Some aspects of a new algorithm for Marangoni convection

Eindhoven University of Technology

Faculty of Mechanical Engineering

Division Materials Technology
Subgroup Multiphaseflow with Heat Transfer

Prof. Dr. H.E.H. Meijer Dr. C.W.M. van der Geld

Some aspects of a new algorithm for Marangoni convection

W.W.F. Pijnappel and C.W.M. van der Geld

Rapportnummer: WFW 96.101

Date : april 1996

Projectname : **Heat transfer enhancement by Marangoni stimulated flow**

Den Dolech 2
P.O. Box 513
5600 MB Eindhoven
Telephone [{..31|0}40] - Voice 2472923, Fax 2447355

Some aspects of a new algorithm for Marangoni convection

W.W.F. Pijnappel
C.W.M. van der Geld

Faculty of Mechanical Engineering
Division Materials Technology
Subgroup Two-Phase Flow with Heat Transfer
Reportnumber WFW 96.101

Introduction

This report describes specialties of a numerical model of surface tension driven convection. The physical model, choice of dimensionless quantities, applications have extensively described elsewhere (van der Geld, den Boer and Pijnappel, "Tests of a numerical model for Marangoni driven convection", Conference Paper Meeting European Two-Phase Flow Group F4, pp. 24, 1994) and are only briefly summarized. The main characteristics of the model are

- the capacity of predicting dynamic deformation of a gas-liquid interface as a consequence of surface tension gradients and
- the basis of spectral analysis and the collocation method.

Emphasis is laid in this report on a careful description of

- the dedicated SVD methods and its implementations
- the alternative ways to predict the interface deformation
- the alternative ways to expand the velocity field
- the coupling to the source code.

This report is intended for future users of the 'Marangoni' software and for those interested in the type of numerical algorithms employed. It highlights the contributions of dr. W. Pijnappel that will only partly be covered by the thesis of A. den Boer (to appear in 1996). For ease of reference extensive appendices have been added.

Contents

Introduction 1

1 Marangoni convection **2**

1.1 Governing equations 2

1.2 Dimensionless equations 3

1.3 The model functions of u and h 5

2 Contour Integration **7**

2.1 Self-adapting grid 7

2.2 Integration 7

2.2.1 Numerical integration along paths with fixed c-value 8

2.2.2 Integration with constant x-values 9

2.2.3 Contour integrals 9

2.3 Storage of Data 10

2.4 The use of the tangential stress equation 11

2.4.1 Time-differentiated tangential stress condition 11

3 Integration Paths **12**

3.1 Integration contour selection 12

3.2 Computation time reduction 14

3.3 The evolution of the height 14

4 Least Squares solution **16**

4.1 Solving Navier-Stokes and the stress conditions simultaneously 17

4.2 The weighting of the tangential stress condition and the contour integrals . 19

4.2.1 The selection procedure of r and t 20

5 Model functions **23**

5.1 Problem definition 23

5.2 Expansions 23

5.3 The liquid height 24

5.4 Time evolution of the height 25

6 Towards a global velocity function with fewer parameters **26**

7 Tests and results 29
7.1 Conclusions 37

A Derivation of the tangential stress condition 40

B The kinematic boundary condition 42

C Transformation of coordinates 44

D A model for the horizontal velocity component satifisfying the no-slip condition at three boundaries 45

E The treatment of the kinematic boundary condition for the case of sine expansion with one linear term 48

F The treatment of the kinematic boundary condition for the case the velocities are expanded in Chebyshev polynomials 50

G A model for the horizontal velocity component satifisfying the no-slip condition at three boundaries 53

H Code of programme 56

Chapter 1

Marangoni convection

Stress gradients at the interface of a fluid may set the fluid into motion. This phenomenon is called Marangoni convection. The origin of the surface stress gradients is in temperature differences at the interface (the thermocapillary effect) or in differences in fluid concentrations (the destillocapilary effect). The liquid flows invoke pressure gradients making the surface to be deformed.

1.1 Governing equations

The objective of the model presented in this paper is to describe the velocity field and the height of the fluid as a function of properties of the fluid, time and surface stress gradients. The first step is to formulate the governing equations.

The following assumptions are made:

1. Flow is 2-D in a Cartesian (x,y)-system of coordinates
2. The surface stress gradients are known at all instants of time
3. The composition of the fluid is homogeneous, with the possible exception of the fluid interface
4. Gravitation is neglected
5. The fluid is incompressable and viscous
6. The dynamic viscosity of the gas above the liquid surface is much less than the dynamic viscosity μ of the fluid

With these assumptions the following governing equations hold

Continuity:

$$\frac{\partial v}{\partial y} = -\frac{\partial u}{\partial x} \tag{1.1}$$

The Navier-Stokes equation in x-direction:

$$\frac{\partial p}{\partial x} = \eta\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) \tag{1.2}$$

The Navier-Stokes equation in y-direction:

$$\frac{\partial p}{\partial y} = \eta\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \rho\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) \tag{1.3}$$

Define N by

$$N = \sqrt{1 + \left(\frac{\partial h}{\partial x}\right)^2} \tag{1.4}$$

The tangential stress condition at the interface:

$$\mu\left(1 - \left(\frac{\partial h}{\partial x}\right)^2\right)\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) + 4\mu\frac{\partial h}{\partial x}\frac{\partial v}{\partial y} = N\left(\frac{\partial \sigma}{\partial x} + \frac{\partial h}{\partial x}\frac{\partial \sigma}{\partial y}\right) \tag{1.5}$$

The normal stress condition at the interface:

$$p = p_0 - \mu\frac{\partial h}{\partial x}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) - \frac{\sigma}{N^3}\frac{\partial^2 h}{\partial x^2} + 2\mu\frac{\partial v}{\partial y} - \frac{1}{N}\frac{\partial h}{\partial x}\left(\frac{\partial \sigma}{\partial x} + \frac{\partial h}{\partial x}\frac{\partial \sigma}{\partial y}\right) \tag{1.6}$$

The kinematic boundary condition at the interface:

$$\frac{\partial h}{\partial t} = v - u\frac{\partial h}{\partial x} \tag{1.7}$$

For the derivation of the stress conditions and of the kinematic boundary condition see appendix A and B.

1.2 Dimensionless equations

The physical model is a cross-section of a rectangular tank containing a thin liquid film. The width of the tank is $2L$ and the height of the liquid, in case the liquid is in rest, is h_0 at all places. We suppose that the Marangoni flow pattern shows a symmetric picture with respect to the center of the tank The bottom of the liquid layer is at $y = 0$ and the walls are at $x = -L$ and $x = L$, see Fig. 1.1. Suppose that $\frac{\partial \sigma}{\partial y} = 0$ and define

$$S_1 = \frac{L}{\Delta\sigma}\frac{\partial \sigma}{\partial x} \tag{1.8}$$

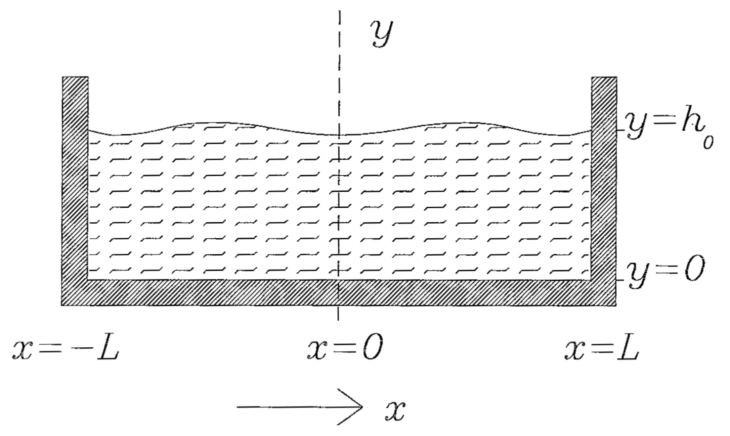


Figure 1.1: Configuration of a thin layer. The flow is symmetric around $x = 0$.

$\Delta\sigma$ is a typical surface tension difference. The characteristic velocity is derived from $\frac{\Delta\sigma}{L} = -\mu\frac{\partial u}{\partial y} \approx -\mu\frac{\Delta u}{\Delta y} \approx -\mu\frac{u-0}{h} = -\mu\frac{u}{h}$. The Reynolds number is $Re = \frac{\Delta\sigma \cdot A \cdot h_0 \cdot \rho}{\mu^2}$ with the aspect ratio $A = \frac{h_0}{L}$.

The governing equations are made dimensionless by the following substitutions:

$$x \rightarrow Lx \tag{1.9}$$
$$y \rightarrow h_0 y \tag{1.10}$$
$$u \rightarrow \frac{\Delta\sigma \cdot h_0}{\mu L} u \tag{1.11}$$
$$h \rightarrow h_0 h \tag{1.12}$$
$$v \rightarrow \frac{\Delta\sigma \cdot {h_0}^2}{\mu L^2} v \tag{1.13}$$
$$t \rightarrow \frac{\mu L^2}{\Delta\sigma \cdot h_0} t \tag{1.14}$$
$$p \rightarrow \frac{\Delta\sigma}{h} p \tag{1.15}$$
$$\sigma \rightarrow \Delta\sigma S_{gr} \tag{1.16}$$

In dimensionless form the governing differential equations read

1. The continuity equation:

$$\frac{\partial v}{\partial y} = -\frac{\partial u}{\partial x} \tag{1.17}$$

2. The Navier-Stokes equation in x-direction:

$$\frac{\partial p}{\partial x} = \frac{\partial^2 u}{\partial y^2} + A^2 \cdot \frac{\partial^2 u}{\partial x^2} - Re \cdot \ A \left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} \right) \tag{1.18}$$

3. The Navier-Stokes equation in y-direction:

$$\begin{aligned} \frac{\partial p}{\partial y} &= A^2 \cdot \frac{\partial^2 v}{\partial y^2} + A^4 \cdot \frac{\partial^2 v}{\partial x^2} - ReA^3 \left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} \right) \\ (D &= A \cdot \frac{\partial h}{\partial x}) \end{aligned} \tag{1.19}$$

4. The tangential stress condition at the interface

$$\left(1 - D^2\right) \left(\frac{\partial u}{\partial y} + A^2 \frac{\partial v}{\partial x} \right) + 4AD\frac{\partial v}{\partial y} = \left(1 + D^2\right) S_1 \tag{1.20}$$

5. The normal stress condition at the interface

$$p = p_0 - A^2 \left(S_{gr}\frac{\partial^2 h}{\partial x^2} \left(1 + D^2\right)^{-1.5} + 2\left(1 - D^2\right)^{-1} \left(\left(1 + D^2\right) \frac{\partial u}{\partial x} + S_1\frac{\partial h}{\partial x} \right) \right) \tag{1.21}$$

1.3 The model functions of u and h

Roll-cells are characteristic for Marangoni flows. This is accounted for in the formula for the horizontal velocity component u of the fluid by describing u as a sum of sines:

$$u = u(x,y) = \sum_{i=0}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} d_{i,k}(t) \cdot \sin(\pi(k+1)x) \cdot \left(\frac{y}{h(x,t)} \right)^{i+1} \tag{1.22}$$

Note: The bounding values $\mathtt{nic} - 1$ and $\mathtt{nkc} - 1$ look somewhat odd. However, such boundaries are normal in the programming language $\mathtt{C}^{++}$, in which for example 'int $a[3]$' declares the variables $a[0]$,$a[1]$ and $a[2]$.

The benefit of this model equation for u is that it satisfies a number of requirements. E.g. the no-slip conditions at the bottom $u(x,0,t) = 0$ and $v(x,0,t) = 0$.
$u(x,y,t) = -u(-x,y,t)$ (u is odd in x)
$u(1,y,t) = 0$

The no-slip condition at the wall demands $v(1,y,t) = 0$. This is satisfied if $d_{i,nkc-1}$ is selected such that for all times, t

$$d_{i,\mathtt{nkc}-1}(t) = \frac{1}{\mathtt{nkc}} \cdot \sum_{k=0}^{\mathtt{nkc}-2} (k+1) \cdot d_{i,k}(t) \cdot (-1)^{\mathtt{nkc}+k} \tag{1.23}$$

In Appendix D a model is given for the horizontal velocity component which implicitly satisfies all aforementioned no-slip conditions and the oddness of the velocity function. The interface function h is defined by

$$h = h(x,t) = \sum_{m=0}^{\mathtt{Ncoef\_b}} b_m(t) \cdot \cos(\pi m x) \tag{1.24}$$

If the liquid is at rest $b_0 = 1$ and $b_m = 0$ for $1 \leq m \leq$ `Ncoef_b`.
$h(1,t) = 1$ because of the no-slip condition at the right wall, i.e.:

$$b_{\mathtt{Ncoef\_b}} = \sum_{m=1}^{\mathtt{Ncoef\_b}-1} b_m(-1)^{m+1+\mathtt{Ncoef\_b}} \tag{1.25}$$

Note that the function h is symmetric with respect to $x = 0$, i.e. $h(-x,t) = h(x,t)$. Other model functions for the horizontal velocity component and the height are discussed in chapter 5.

Chapter 2

Contour Integration

In this chapter we discuss the algorithm as it is programmed in $\mathtt{C}^{++}$. The heart of the program is the routine Timestep ::ContourInt, in which contour integrals are calculated.

2.1 Self-adapting grid

For each liquid particle having coordinates (x, y) is $-1 \leq x \leq 1$ and $0 \leq y \leq h(x,t)$. Because of symmetry it is sufficient to study the motion of the liquid particles with $0 \leq x \leq 1$, see section 1.2. With the help of the coordinate transformation $c = \frac{y}{h(x)}$ the coordinates of liquid particles are represented in the x-c coordinate system, in which $0 \leq x \leq 1$ and $0 \leq c \leq 1$. In the x-c system `ni` `nk` fitting point are chosen: $(x[k], c[i])$, with $0 \leq k <$ `nk` and $0 \leq i <$ `ni`. These point are chosen equidistant or according to some other partition, e.g. in the collocation method: the x- and c-coordinates are the zeros of the (shifted) Legendre polynomials [2]. These points constitute a self-adapting grid system and are somewhat misleadingly called fitting points; They are the begin- and endpoints of integration paths.

2.2 Integration

To each pair of fitting points with the same c-coordinate, say (x_1, c) and (x_2, c), corresponds an integration path along straight lines: from (x_1, c) to (x_2, c); next from (x_2, c) to $(x_2, 1+\epsilon)$ with $\epsilon > 0$ infinitesimal (leaving the liquid); then from $(x_2, 1+\epsilon)$ to $(x_1, 1+\epsilon)$ and then back from $(x_1, 1+\epsilon)$ to (x_1, c). For the algorithm the following parts of the path are of importance: The path from (x_1, c) to (x_2, c); the path from (x_2, c) to $(x_2, 1)$; then from $(x_2, 1-\epsilon)$ to $(x_2, 1+\epsilon)$ with $\epsilon > 0$ infinitesimal (leaving the liquid; use the normal stress condition); then from $(x_1, 1+\epsilon)$ to $(x_1, 1-\epsilon)$ (entering the liquid); and finally from $(x_1, 1)$ to (x_1, c) .

The difference in pressure between the fitting points (x_1, c) and (x_2, c) is

$$p(x_2,c) - p(x_1,c) = \int_{x_1}^{x_2} \frac{\partial p}{\partial s} ds = \int_{x_1}^{x_2} \left(\frac{\partial p}{\partial x} + c \frac{\partial h}{\partial x} \frac{\partial p}{\partial y} \right) dx \tag{2.1}$$

Here s is the x-coordinate in the x-c system. For the derivation see appendix C.

Between (x_2, c) and $(x_2, 1)$ the difference in pressure is

$$p(x_2,1) - p(x_2,c) = \int_c^1 \frac{\partial p}{\partial c} dc = \int_{ch(x_2)}^{h(x_2)} \frac{\partial p}{\partial y} dy \tag{2.2}$$

If the velocity field, the height and the gradients S_1 and S_{gr} (see section 1.2) are known at a time, then every contour integral supplies with the help of the Navier-Stokes equations a linear equation with as unknowns the time-derivatives of the coefficients $d_{ik}(t)$ and $b_m(t)$ ($0 \leq i <$ `nic` ; $0 \leq k <$ `nkc`; $0 \leq b_m \leq$ `Ncoef_b`).

The time-derivatives of the coefficients $b_m(t)$ are calculated using the kinematic boundary condition (see chapter 3 section 3.3; see also routine New_DB::NewDB). Thus every contour integral provides a linear equation in the unknowns e_{ik} $\left(= \frac{\partial d_{ik}}{\partial t}\right)$.

2.2.1 Numerical integration along paths with fixed c-value

In order to compute the pressure difference $p(x_2, c) - p(x_1, c)$ the following integrals have to be calculated:

1) $\int_{x_1}^{x_2} \frac{\partial^2 u}{\partial y^2} dx$ 2) $\int_{x_1}^{x_2} \frac{\partial^2 u}{\partial x^2} dx$ 3) $\int_{x_1}^{x_2} \frac{\partial u}{\partial t} dx$

4) $\int_{x_1}^{x_2} u \frac{\partial u}{\partial x} dx$ 5) $\int_{x_1}^{x_2} v \frac{\partial u}{\partial y} dx$ 6) $\int_{x_1}^{x_2} c \frac{\partial h}{\partial x} \frac{\partial^2 v}{\partial y^2} dx$

7) $\int_{x_1}^{x_2} c \frac{\partial h}{\partial x} \frac{\partial^2 v}{\partial x^2} dx$ 8) $\int_{x_1}^{x_2} c \frac{\partial h}{\partial x} \frac{\partial v}{\partial t} dx$ 9) $\int_{x_1}^{x_2} c \frac{\partial h}{\partial x} u \frac{\partial v}{\partial x} dx$ 10) $\int_{x_1}^{x_2} c \frac{\partial h}{\partial x} v \frac{\partial v}{\partial y} dx$

A total of ten integrals. It is tempting to compute the second integral via $\left[\frac{\partial u}{\partial x}\right]_{x_1}^{x_2}$. But that not correct, because the second integral is $\int_{x_1}^{x_2} \frac{\partial^2 u}{\partial x^2}(x, ch(x), t) dx$ and that is not the same as

$$\left[\frac{\partial u}{\partial x}\right]_{x_1}^{x_2} \left(= \int_{x_1}^{x_2} \frac{\partial^2 u}{\partial x^2}(x,y,t) dx_{|y=ch(x)} \right)$$

Analogously integral 4 cannot be calculated via $\left[\frac{1}{2}u^2\right]_{x_1}^{x_2}$.

The fourth integral is calculated numerically. For that purpose the interval (x_1, x_2) is partitioned into a number of equidistant points. This number is chosen proportional to the length of the interval. For the step length chosen, 0.001, the numerical error will be small (accuracy in ten decimals) if the third derivative of the integrand does not vary a lot, which is a valid assumption in most applications. For each point of the partition the values for u and $\frac{\partial u}{\partial x}$ are computed. The values of $u\frac{\partial u}{\partial x}$ for all partition points are multiplied by a weight factor and added according to a fourth order integration method. The ratio of the weight factors is 1:4:2:4:...:2:4:1. It is not sensible to bring the fourfold summation of $u\frac{\partial u}{\partial x}$ in front of the integral in order to get simple integrands, for in that case the number of computer operations is too large. The same procedure is followed for the integrals 3,5,7,8,9 and 10.

Another strategy is followed for the numerical integration of the integrals 1,2 and 6 and the integral 8 as far as the computation of the coefficients of e_{ik} goes. The summation marks are placed in front of the integrals as well as all terms which do not depend on x. The remaining integrands are simple functions, which are computed numerically.

2.2.2 Integration with constant x-values

For the computation of $p(x_2, 1) - p(x_2, c)$ the following integrals have to be computed, the integration path being from $ch(x_2)$ to $h(x_2)$:

$$1) \int \frac{\partial^2 v}{\partial y^2} dx \quad 2) \int \frac{\partial^2 v}{\partial x^2} dx \quad 3) \int \frac{\partial v}{\partial t} dx$$

$$4) \int u \frac{\partial v}{\partial x} dx \quad 5) \int v \frac{\partial v}{\partial y} dx$$

The first integral is equal to $\left[\frac{\partial v}{\partial y}\right]_{ch(x_2)}^{h(x_2)}$

The fifth integral is equal to $\left[\frac{1}{2}v^2\right]_{ch(x_2)}^{h(x_2)}$

The third integral does not need not to be solved numerically because of its simple form, see the appendix called Formulae. So only the second and fourth integral are solved numerically. This is done in the way explained in section 2.2.1.

2.2.3 Contour integrals

The number of contours is `ni nk`. Each contour starts at one of the `ni nk` fitting points and goes from there via the curve with constant c-value to the wall, i.e. point $(1, c)$. Then from $(1, c)$ to $(1, 1)$ and from there as explained in section 2.2.

The sequences $x[0], x[1], ..., x[\mathtt{nk} - 1]$ and $c[0], c[1], ..., c[\mathtt{ni} - 1]$ as defined in routine Init_New _Run ::StartNew are ordered from large to small (in subroutine c_Computation ::c_Comp). In order to computate the numerical integrals efficiently the first contour starts at $(x[0], c[0])$, the second at $(x[1], c[0])$ and e.g. the `nk`-th at $(x[\mathtt{nk} - 1], c[0])$. For the calculation of the difference in pressure $p(1, c[0]) - p(x[k], c[0])$ only $p(x[k-1], c[0]) - p(x[k], c[0])$ has to be calculated, because the rest is known from preceding calculations. The step size is practically constant. So, the number of steps in the numerical process and the computing time is proportional to the length of the integration interval.

The calculated values for $p(x[k], 1) - p(x[k], c[0])$ for $k = 0...\mathtt{nk} - 1$ are stored in arrays. After the calculation of the contour integral with starting point $(x[\mathtt{nk} - 1], c[0])$ we continue with the contour integral with starting point $(x[0], c[1])$. For this contour the integral $p(x[0], 1) - p(x[0], c[0])$ is known, and is used in the computation of $p(x[0], 1) - p(x[0], c[1])$. Thus, the fitting points are passed through from right to left and from top to bottom. Other ways to pass through the fitting points are discussed in chapter 3.

The computation of the differences in pressure at the interface where the normal stress condition has to be used is carried out in routine TimeStep ::FactorCalculation.

Differences in pressure at the wall $(x = 1)$ are computed in TimeStep ::ContourInt. The computations at the wall are much simpler than the computations within the liquid $(0 < x < 1)$ because of the no-slip conditions. E.g.: Of all the integrals of section 2.2.2 only the second one is not zero.

2.3 Storage of Data

The computation of a contour integral results in a linear equation with unknowns e_{ik} with $0 \leq i < \texttt{nic}$ and $0 \leq k < \texttt{nkc}$. The unknowns $e_{i,\texttt{nkc}-1}, (0 \leq i < \texttt{nic})$, are expressed as linear combinations of the remaining unknowns (see section 1.3), so that in the end there is a linear equation in the unknowns e_{ik} with $0 \leq i < \texttt{nic}$ and $0 \leq k < \texttt{nkc} - 1$. The coefficients are stored in the array helpa In vector notation $helpa_1^T e = rm_1$ with rm_1 ('right member') a scalar. For the r-th contour integral $helpa_r^T e = rm_r$. This yields

$$\begin{pmatrix} helpa_1^T \\ helpa_2^T \\ \vdots \\ helpa_{\texttt{ni}\,\texttt{nk}}^T \end{pmatrix} e = \begin{pmatrix} rm_1 \\ rm_2 \\ \vdots \\ rm_{\texttt{ni}\,\texttt{nk}} \end{pmatrix}$$

This system is overdetermined, so a least squares solution is obtained. To that end both members of the matrix equation are multiplied by the transpose of the left matrix, i.e. $(helpa_1, ..., helpa_{\texttt{ni}\,\texttt{nk}})$. The resulting matrix equation generally has a unique solution since its matrix is square.

This matrix equation is written in the following way

$$(helpa_1 \cdot helpa_1^T + ... + helpa_{\texttt{ni}\,\texttt{nk}} \cdot helpa_{\texttt{ni}\,\texttt{nk}}^T)e = rm_1 \cdot helpa_1 + ... + rm_{\texttt{ni}\,\texttt{nk}} \cdot helpa_{\texttt{ni}\,\texttt{nk}}$$

This notation makes clear that this equation can be made up bit by bit when the integrals are calculated. After the computation of the first contour integral $helpa_1$ and rm_1 are known. With this $ah = helpa_1 \cdot helpa_1^T$ and $bvec = rm_1 \cdot helpa_1$ are calculated. After the computation of the second contour integral $helpa_2$ and rm_2 are known. In fact, the array helpa with contents $helpa_1$ is overwritten by $helpa_2$. $helpa_2 \cdot helpa_2^T$ is added to matrix ah and $rm_2 \cdot helpa_2$ to vector bvec. And so on.

The matrix $(helpa_1, ..., helpa_{\texttt{ni}\,\texttt{nk}})$ contains $\texttt{nic}(\texttt{nkc} - 1)\texttt{ni}\ \texttt{nk}$ elements. This number exceeds the capacity of a personal computer for large values of `ni` and `nk`. Owing to the strategy followed the matrix ah is independent of `ni` and `nk` and at all times only one column (not always the same) of the large matrix $(helpa_1, ..., helpa_{\texttt{ni}\,\texttt{nk}})$ has been stored.

2.4 The use of the tangential stress equation

Unlike the normal stress condition, the tangential stress condition is not used in the contour integrations. The tangential stress condition is used in SolveWTT_Cond ::Solve_E_WithTangTens_Cond.

2.4.1 Time-differentiated tangential stress condition

Solving the matrix equation resulting from the contour integrals at time t yields the coefficients e_{ik} $(= \frac{\partial d_{ik}}{\partial t})$, see section 2.3. Time-integration of the e_{ik} yields the coefficients d_{ik} and so the velocity field at time $t + \Delta t$; and time-integration of the kinematic boundary condition, see 1.7 yields the surface profile at time $t + \Delta t$. The tangential stress condition has not been used so far. So at first sight this equation might seem to be redundant. However, the tangential stress condition is the driving force and therefore the most important equation. In chapter 4 it is shown how to do justice to the important role of this condition. In this section it is shown how the incorporation of the tangential stress condition was done in a first attempt. The resulting equations are still present in the algorithms, but do not play an important role any more. They might be deleted, if preferred, without effecting the resulting output appreciably.

The formula for the tangential stress condition is differentiated with respect to time. This yields a linear equation in the unknowns e_{ik} for each point $(x[k], h(x[k]))$ of the fluid interface. This equation is added to the equations obtained from the contour integrals.

Use is made of weight factors, since there are `ni` `nk` contour paths and only `nk` equations corresponding to tangential stress conditions. First every contour equation is normalized 1 such that the largest coefficient of every equation (in absolute value) is 1. This is also done for every equation obtained from tangential stress conditions. Next, each of the lastmentioned equations is multiplied by `ni`. In this way the tangential stress equations and the contour equations become equally important in the solution of the matrix equation.

Chapter 3

Integration Paths

3.1 Integration contour selection

With the aid of the Navier-Stokes equations the derivatives of the liquid pressure, $\frac{\partial p}{\partial x}$ and $\frac{\partial p}{\partial y}$, are expressed in terms of velocities and their derivatives, see 1.2 and 1.3. The liquid velocities are linearly expanded in the parameters d_{ik} $(0 \leq i < \texttt{nic}\ ;\ 0 \leq k < \texttt{nkc})$. Thus $\frac{\partial p}{\partial x}$ and $\frac{\partial p}{\partial y}$ are linear expressions in the parameters d_{ik} and their time derivatives $e_{ik}(= \frac{\partial d_{ik}}{\partial t})$. At any time the parameters d_{ik} are known and the parameters e_{ik} are the unknowns in the routine ContourInt.

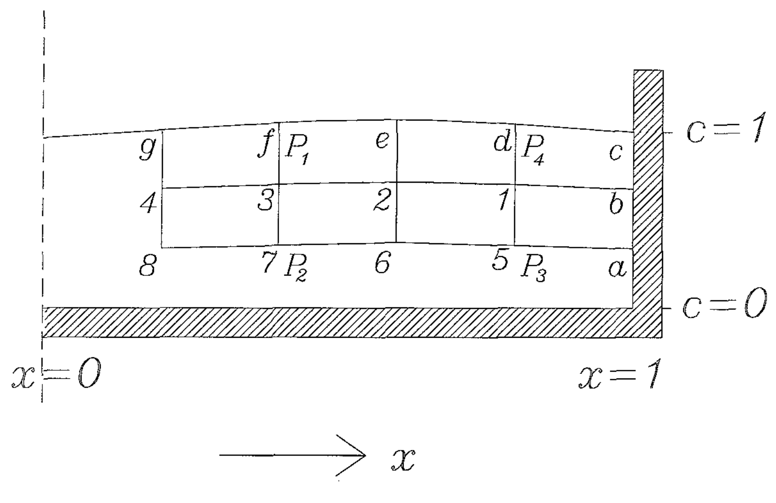


Figure 3.1: Paths of integration in the cavity.

In Figure 3.1 p_0 is the air pressure, p_1 and p_4 are pressures at two different points, P_1 and P_4 respectively, near the interface, just below the liquid surface; p_2 and p_3 are pressures at two points in the liquid. The path between P_1 and P_2 and the path between P_3 and P_4 are parallel to the sidewalls.

The path between P_2 and P_3 is not always straight but follows the height of the liquid surface. There is a number, c, such that for every point (x', y') of the path, $y' = c\, h(x')$, $h(x')$ being the liquid height at $x = x'$. It is obvious that

$$p_0 - p_4 + \sum_{i=0}^{3} (p_{i+1} - p_i) = 0$$

The terms $p_1 - p_0$ and $p_0 - p_4$ are computed with the normal stress condition. They are independent of the unknowns e_{ik}. The remaining terms are computed by integration along their paths. E.g.

$$p_2 - p_1 = p(x, c\, h(x)) - p(x, h(x)) = \int_{h(x)}^{c\, h(x)} \frac{\partial p}{\partial y} dy$$

To illustrate the contours used in version 1 of the algorithm examples are given with `nk` = 4 and `ni` = 2 (see Figure 3.1). The first contour is *air* − *d* − 1 − *b* − *c* − *air*,
The second contour is *air* − *e* − 2 − *b* − *c* − *air*.
The third contour is *air* − *f* − 3 − *b* − *c* − *air*.
and so on. The second contour can also be described as *air* − *e* − 2 − 1 − *b* − *c* − *air*, which shows that after the calculation of the first contour only the paths *air* − *e*, *e* − 2 and 2 − 1 need to be calculated. And for the computation of the last contour, *air* − *g* − 8 − *a* − *c* − *air*, only the paths 4 − 8 and 8 − 7 need to be calculated.

A disadvantage of this strategy is that data obtained from the upper right part of the liquid has larger weight in the least squares solution method than other parts of the liquid, whereas the largest velocities occur in the upper left corner of the liquid. E.g. In the example of Figure 3.1 the path (1 − *b*) is used in half the total number of contour equations, and path (*b* − *c*) in all equations. The velocities at point *b* are however low, so in all these equations the path (*b* − *c*) does not contribute much to the fit of the model parameters.

The contour equations are an overdetermined system which is solved in the least squares sense. It is important to give the greatest weight to integration parts with the largest velocities, since a least squares procedure fits best the data with the largest weigt.

One way of doing this is to start the first contour at 4 (version 2), thus the first contour is *air* − *h* − *i* − 4 − *g* − *air* (See Figure 3.1). The second contour then should be *air* − *h* − *i* − 3 − *f* − *air* etc. In this way the path *h* − *i* appears in every contour, and the side wall with its no-slip conditions does not appear in any of the contours.

In another version (version 3) of the Marangoni program we follow the next strategy. The first contour is *air* − *d* − 1 − *b* − *c* − *air* (See Figure 2). The second contour is *air* − *e* − 2 − 1 − *d* − *air* and the third *air* − *f* − 3 − 2 − *e* − *air*, and so on. In this way all data on a path of constant c-value is weighted equally. This version is preferable when the velocities are not only large in the upper left part of the liquid. The computational burden is the same in all strategies mentioned.

3.2 Computation time reduction

The essential body of the Marangoni algorithm comprises the computations of the contour integrals. The numerical calculations of these integrals usually are the most time consuming parts of the algorithm. The composing of the coefficient matrix *ah* in routine ContourInt is, however, of the same order of magnitude. The composing of the matrix *ah* and vector *bvec* was done with the following programming code (mx is a normalizing constant):

```
for ( mx*=mx , k=0 ; k < (nkc-1)*nic ; k++ ){
  bvec[k] += rm * helpa[k]/mx;
  for ( i=0 ; i < nic*(nkc-1) ; i++ )
    ah[k][i] += helpa[k] * helpa[i]/mx;
}
```

Another way of writing this code is:

```
for ( k=0 ; k < (nkc-1)*nic ; k++) helpa[k] /= mx;
for ( rm /=mx , k=0 ; k < (nkc-1)*nic ; k++ ){
  bvec[k] += rm * helpa[k];
  for ( i=k ; i < nic*(nkc-1) ; i++ )
    ah[k][i] += helpa[k] * helpa[i];
}
```

The difference between these two pieces of program is that in the first code the number of multiplications and divisions of double precision numbers is $2\,\mathtt{nic}(\mathtt{nkc}-1)(\mathtt{nic}(\mathtt{nkc}-1)+1)$ and in the second $\frac{1}{2}\,\mathtt{nic}(\mathtt{nkc}-1)(\mathtt{nic}(\mathtt{nkc}-1)+1)+2\,\mathtt{nic}(\mathtt{nkc}-1)+1$, which is a reduction in computing time of almost a factor 4. By using the second option, a considerable reduction in the computation time of the program is therefore achieved.

The computation time of the numerical integrations is kept as low as possible. The calculation of succeeding sine and cosine functions is done with the sum rules for sines and cosines, and the calculation of succeeding powers is done without use of the time consuming standard pow-function, but with simple additions and multiplications.

3.3 The evolution of the height

The kinematic boundary condition, see 1.7 is used for the computation of the coefficients b_m $(1 \leq m \leq \mathtt{Ncoef\_b})$ of the height function, h. This can be done in two ways.

In the first way, first the derivatives $\frac{\partial h}{\partial t}$ and $\frac{\partial^2 h}{\partial x \partial t}$ are computed at time t_0, $\frac{\partial^2 h}{\partial x \partial t}$ by differentiating $\frac{\partial h}{\partial t}$ as given by 1.7. The functions h and $\frac{\partial h}{\partial x}$ at time $t + \Delta t$ are calculated by time-integration of the derivatives $\frac{\partial h}{\partial t}$ and $\frac{\partial^2 h}{\partial x \partial t}$ according to a second order Adams-Bashford formula. The values for h and $\frac{\partial h}{\partial x}$ at time $t + \Delta t$ are computed at nk points on the surface. The functions h and $\frac{\partial h}{\partial x}$ can also be expressed as a linear combination of the parameters

b_m at time $t + \Delta t$, which have to be determined. A matrix equation results with vector b_m unknown, The matrix has size 2`nk` by `Ncoef_b`. Since `Ncoef_b` is taken less than 2 `nk` the values of b_m can be computed from this equation.

Another way of determining the parameters b_m is by equalizing the coefficients of corresponding sines and cosines on both sides of the equation $\frac{\partial h}{\partial t} = v - u\,\frac{\partial h}{\partial x}$. Note that in this equation all constituents, i.e. $\frac{\partial h}{\partial t}, v, u$ and $\frac{\partial h}{\partial x}$ are linear sums of sines and/or cosines.

Values of $\frac{\partial b_m}{\partial t}$ are derived from these equalities. Again, an Adams-Bashford formula is used, now to obtain the new values for the coefficients b_m from $\frac{\partial b_m}{\partial t}$. A remarkable fact is that the number of unknown height coefficients grows with time, caused by the product $u\,\frac{\partial h}{\partial x}$. This is shown in appendix D.

In most versions of the algorithm this approach is used. It is more accurate than the first one, because the coefficients $\frac{\partial b_m}{\partial t}$ are calculated exactly whereas in the first approach the values for h and $\frac{\partial h}{\partial x}$ are calculated at only `nk` points of the surface. In both cases an Adams-Bashford integration is needed. In routine New_DB ::NewH the code for the described calculations can be found. The number of height coefficients grows linear in time. Calculations show, that if the number of computed b-coefficients is twice the number of height coefficients (`Ncoef_b`) the error made is neglectable. (See appendix D.)

Chapter 4

Least Squares solution

The Navier-Stokes equations and the normal stress condition at the interface, see eqs. 1.18, 1.19 and 1.21 are used in every version of the Marangoni program in routine `TimeStep::ContourInt` for the computation of contour integrals, see section 2.2. Each fitting point corresponds to one contour integral. The number of fitting points (= `ni` `nk`) must be very large to get accurate results for the parameters of the velocity functions. In many program tests the number of fitting points selected was 10000 . The calculation of the contour integrals results in a matrix equation $Ae = b$ in which the coefficient matrix A then is of size 10000 by `nic` (`nkc` -1). Note that `nic` (`nkc` -1) is the number of unknowns e_{ik}. Matrices of this size are very large for personal computers.

A trick has been used to reduce the number of matrix entries: For every fitting point the coefficients of the linear equation of the corresponding contour integral are calculated.

$$helpa^T\ e = rm \tag{4.1}$$

Here helpa is the (column)vector of coefficients, e is the vector of unknowns e_{ik} and rm is the constant of the equation. Both members of equation 4.1 are multiplied by helpa, yielding $helpa\ helpa^T\ e = rm\ helpa$. Matrix $helpa\ helpa^T$ is added to a matrix called ah, and vector $rm\ helpa$ is added to vector bvec. Initially ah is a zero matrix and bvec is a zero vector. Matrix ah is a square, self-adjoint matrix of size `nic` (`nkc` $-$ 1). This size is considerably less than 1000 by `nic` (`nkc` $-$ 1) which reduces the computer memory necessary. The solution of the matrix equation $ah\ e = bvec$ is the least squares solution of the aforementioned matrix equation $A\ e = b$.

Routine `TimeStep::ContourInt` is not the only routine which provides equations in the unknowns e_{ik}. In routine Solve_E_WithTangTens_Cond the time-differentiated tangential stress condition, see section 2.4.1, provides `nk` extra linear equations. So, in total there are (`ni` $+ 1$) `nk` linear equations in `nik` (`nkc` $-$ 1) unknowns. Solving the system $ah\ e = b$ yields the values of $e_{ik}(= \frac{\partial d_{ik}}{\partial t})$, from which the velocity coefficients d_{ik} can be computed, e.g. by using a second order Adams-Bashford formula.

The tangential stress condition is the most important differential equation in the surface tension gradient driven convection. It encompasses the driving force of the liquid flows.

To fully account for its importance it has to be used a second time. How this was done first, is explained in short in the next few lines. In the next sections it is expounded how the importance of the tangential stress condition could be shown to full advantage.

For each point of the liquid surface the tangential stress condition yields an equation in the variables d_{ik} (see routine NewNkNi ::NewNkNi_uc_h). There are `nk` equations of this kind. There also exists `nic` (`nkc` $-$ 1) equations of the type $d_{ik} = \tilde{d_{ik}}$ in which the LHS is a variable whereas the $\tilde{d_{ik}}$ on the right hand side is the numerical value of the variable d_{ik}. These values $\tilde{d_{ik}}$ have already been calculated in the routine TimeStep::ContourInt, where the contour integrations are exployed.

To achieve a proper balance of the driving surface tension gradients and the flow history and liquid stresses as represented by the Navier Stokes equations and the contour integrals it is mandatory to weight all these equations properly. This could be done (and was originally done) in the following way. Let every equation be divided by the maximum absolute value of its coefficients. Then, divide the first `nk` equations (which are derived from the tangential stress condition) by `nk` and multiply the rest by $\frac{b}{\texttt{nic}(\texttt{nkc}-1)}$. If b would be 1 then the first `nk` and the last `nic` (`nkc` $-$ 1) equations would contribute equally to the solution. But that is not exactly what is required. The tangential stress condition has to be satisfied as exactly as possible, because it encompasses the driving force, and given that demand the contour equations have to be satisfied as accurate as possible. In the following this weighting procedure is treated in detail.

4.1 Solving Navier-Stokes and the stress conditions simultaneously

In the procedure of section 4 the vector e had to be solved from the matrix equation comprising the time-differentiated tangential stress condition at the interface and the contour integrations. After that, the vector d is computed two times over again. First by integrating this vector e, secondly by solving the system formed by the tangential stress condition at the interface and the $\tilde{d}$-values obtained. A better way to handle the computation of $\{d\}$ is the following procedure. An Adams-Bashford formula reads

$$d_{t_0+\Delta t} = d_{t_0} + (\frac{3}{2} e_{t_0} - \frac{1}{2} e_{t_0-\Delta t})\ \Delta t$$

Here $d_{t_0+\Delta t}$ is the value of d at time $t_0 + \Delta t$. At this stage its value is unknown, whereas d_{t_0} and e_{t_0} are known values at time t_0. Thus,

$$e_{t_0} = \frac{2}{3\ \Delta t} d_{t_0+\Delta t} + \frac{1}{3} e_{t_0-\Delta t} - \frac{2\ d_{t_0}}{3\ \Delta t}$$

Substitution into $ah\ e_{t_0} = bvec$, see section 4, results in

$$ah\ d_{t_0+\Delta t} = \frac{3\ \Delta t}{2}\ bvec + ah(d_{t_0} - \frac{1}{2}\ \Delta t\ e_{t_0-\Delta t})$$

In this equation ah is square of size `nic` (`nkc` $-$ 1). It is the product of the transpose of a matrix A of size `ni` `nk` by `nic` (`nkc` $-$ 1) by matrix A itself, i.e. $ah = A^T\ A$, and the right hand member can be written as $A^T\ q$, for a certain vector q of length `nic` (`nkc` $-$ 1).
Thus $A^T\ A\ d = A^T\ q$ and $A\ d = q$.
The objective of this section is to obtain the total least squares solution of this last system. To that end the vector d is extended by the scalar -1. Then the following block matrix equation is formed

$$(A\ \ q)\begin{pmatrix} d \\ -1 \end{pmatrix} = 0$$

I.e. $A'\ d' = 0$ with $A' = (A\ \ q)$ and $d' = \begin{pmatrix} d \\ -1 \end{pmatrix}$

$(A')^T\ A'$ is a positive matrix, so it is symmetric, and has an eigenvalue decomposition, its eigenvalues are real, non-negative and there exists an orthonormal basis of eigenvectors.

So, when x is an eigenvector of $(A')^T\ A'$ with norm 1 and with positive eigenvalue λ^2, then

$$(A')^T\ A'\ x = \lambda^2\ x$$

Define $y = \frac{A'\ x}{\lambda}$, then

$$\lambda^2\ x = (A')^T\ A'\ x = \lambda\ (A')^T\ y$$

and

$$A'\ (A')^T\ y = \lambda\ A'\ x = \lambda^2\ y$$

So, y is an eigenvector of matrix $A'\ (A')^T$. Since $y = \lambda^{-1} A'\ x$, $y^T = \lambda^{-1} x^T A'^T$ and

$$\lambda^2\ y^T\ y = x^T\ (A')^T\ A'\ x = \lambda^2\ x^T\ x = \lambda^2$$

since x has norm 1. Hence $y^T\ y = 1$, so y has norm 1.

Define $n =$ `nkc`(`nic` $-$ 1) $+ 1$. Order eigenvectors $x_1, x_2, \cdots, x_n$ and $y_1, y_2, \cdots, y_n$ such that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$

$$A' = \lambda_1\ y_1\ x_1^T + \cdots + \lambda_n\ y_n\ x_n^T$$

which can be verified by multiplying both sides with the elements of the basis, i.e. the individual eigenvectors $x_1, x_2, \cdots, x_n$.

The matrix $\tilde{A}$ of rank r for which the Frobenius norm (or the l_2 norm) of $A' - \tilde{A}$ is minimal is

$$\tilde{A} = \lambda_1\ y_1\ x_1^T + \lambda_2\ y_2\ x_2^T + \cdots + \lambda_r\ y_r\ x_r^T \tag{4.2}$$

which is a well known property of the singular values λ_j [3]. The choice for r and the purpose of this choice is discussed later on. To obtain a solution of $\tilde{A}\ d' = 0$, see eq. 4.2 above, the matrix $\tilde{A}$ is written as a block matrix $\tilde{A} = (A''\ \ q'')$, where q'' is a vector of length `ni` $\cdot$ `nk`. For $1 \leq i \leq r$ let the eigenvectors x_i be:

$$x_i = \begin{pmatrix} x_i'' \\ s_i \end{pmatrix}$$

The equation $\tilde{A}\; d' = 0$ yields

$$(A''\; q'')\begin{pmatrix} d \\ -1 \end{pmatrix} = 0 \qquad \textit{or} \qquad A''\; d = q''$$

Eq. 4.2 yields

$$A''\; d = \left(\lambda_1\; y_1\; (x_1'')^T + \cdots + \lambda_r\; y_r\; (x_r'')^T\right)\; d$$

and

$$q'' = \lambda_1\; y_1\; s_1 + \cdots + \lambda_r\; y_r\; s_r$$

Because of $A''d = q''$ and $\lambda y = A'x$ this yields

$$\left(A'\; x_1(x'')^T + \cdots + A'x_r(x'')^T\right)\; d = s_1\; A'\; x_1 + \cdots + s_r\; A'\; x_r$$

$A''d = q''$, so:

$$(A'')^T A''d = (A'')^T q''$$

Substitution for A'' an q'' shows

$$\left(x_1''\; x_1^T\; A'^T + \cdots x_r''\; x_r^T\; A'^T\right)\; \left(A'\; x_1\; (x_1'')^T + \cdots + A'\; x_r\; (x_r'')^T\right)\; d$$

$$= \left(x_1''\; x_1^T\; A'^T + \cdots x_r''\; x_r^T\; A'^T\right)\; (s_1\; A'\; x_1 + \cdots + s_r\; A'\; x_r)$$

Note that $A'^T\; A'\; x = \lambda^2\; x$. Therefore

$$\left(x_1''\; x_1^T + \cdots x_r''\; x_r^T\right)\; \left(\lambda_1^2\; x_1\; (x_1'')^T + \cdots + \lambda_r^2\; x_r\; (x_r'')^T\right)\; d$$

$$= \left(x_1''\; x_1^T + \cdots x_r''\; x_r^T\right)\; (s_1\; \lambda_1^2\; x_1 + \cdots + s_r\; \lambda_r^2\; x_r)$$

The eigenvectors $x_1, \cdots, x_r$ form an orthonormal set, whence

$$\left(\lambda_1^2\; x_1''\; (x_1'')^T + \cdots + \lambda_r^2\; x_r''\; (x_r'')^T\right)\; d = s_1\; \lambda_1^2\; x_1'' + \cdots + s_r\; \lambda_r^2\; x_r''$$

At the end of the next section these equations are combined with the equations obtained from the tangential stress condition.

4.2 The weighting of the tangential stress condition and the contour integrals

The tangential stress condition at the interface yields `nk` equations in the unknowns d_{ik}. The `nk` equations are written in matrix form: $B\; d = p$, or in block matrix form

$$B'\; d' = 0 \texttt{with}\; B' = (B\; p) \texttt{ and }\; d' = \begin{pmatrix} d \\ -1 \end{pmatrix}$$

B' is an augmented matrix of size `nk` by `nkc` (`nic` $-$ 1) + 1. Like in section 4.1 the matrices $\tilde{B}$ and B'' are defined in order to obtain well-conditioned matrix equations. If the eigenvalues of $(B')^T\ B'$ are $u_1, u_2, \cdots, u_n$ ($u_i = \begin{pmatrix} u_i'' \\ l_i \end{pmatrix}$) and the corresponding eigenvalues are $\mu_1^2, \cdots, \mu_n^2$ then it is derived in a similar manner that

$$\left(\mu_1^2\ u_1''\ (u_1'')^T + \cdots + \mu_t^2\ u_t''\ (u_t'')^T\right)\ d = l_1\ \mu_1^2\ u_1'' + \cdots + l_t\ \mu_t^2\ u_t''$$

with t = rank $\tilde{B}$ In the next section this equation is combined with the end equation of section 4.1. The resulting matrix equation is solved.

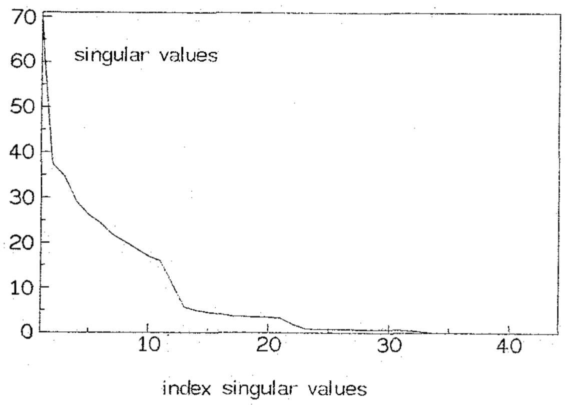


Figure 4.1: Characteristic plot of the singular values of matrix A

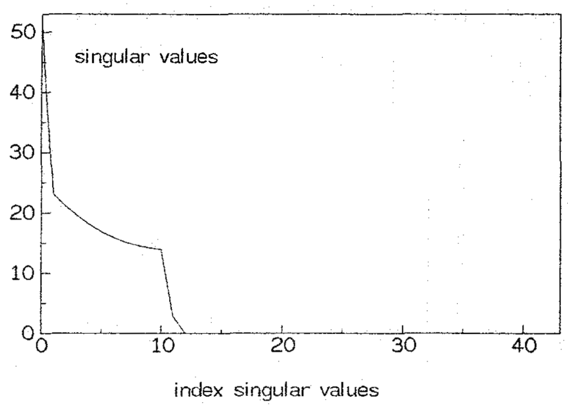


Figure 4.2: Characteristic plot of the singular values of matrix B

4.2.1 The selection procedure of r and t

The singular values of B show a characteristic pattern, see Fig. 4.2. There is a sharp transition from significant singular values to noise related singular values. Let t be the number of significant singular values. In the example of Figure 4.2 `nkc` = 12 and `nic` = 4.

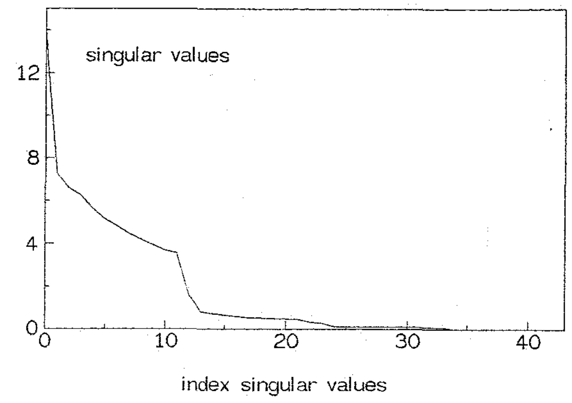


Figure 4.3: Characteristic plot of the singular values of matrix C

There are eleven (`nkc` -1) or twelve (`nkc`) significant singular values. The twelfth singular value is very small compared to the first eleven. Not much information would be lost if t would be set 11 rather than 12.

Let us put $r = n - t - 1$ and construct the following matrix equation

$$\begin{pmatrix} \tilde{A} \\ \tilde{B} \end{pmatrix} d' = 0$$

in which $\tilde{A}$ has rank t and $\tilde{B}$ has rank r. Thus, the rank of $\begin{pmatrix} \tilde{A} \\ \tilde{B} \end{pmatrix}$ is $n-1$ or less. In practice the rank is usually $n-1$ and $\begin{pmatrix} A'' \\ B'' \end{pmatrix}$ has full rank. In this case the least squares solution of the following matrix equation is an exact solution.

$$\begin{pmatrix} A'' \\ B'' \end{pmatrix} d = \begin{pmatrix} q'' \\ p'' \end{pmatrix}$$

the So, the required solution for d is computed from

$$(A'')^T\, A''\, d + (B'')^T\, B''\, d = A^T\, q'' + B^T\, p''$$

The advantage of this approach is that the d-values computed in this way meet almost exactly the tangential stress conditions at minimal loss of accuracy in the contour integrals.

Figure 4.1 shows a characteristic plot of the singular values of matrix A. The corresponding table 4.1, column 2, shows jumps in the graph at index 11, 21 an 31. For $t = 12$, $r = n - t - 1 = (12 * 3 + 1) - 12 - 1 = 24$.

Matrix $(A'')^T\, A'' + (B'')^T\, B''$ is the sum of semi-definite matrices; so it is positive. It is non-singular so it is positive definite and can be written as $C^T\, C$ for a certain non-singular matrix C. The singular value plot of matrix C is shown in Figure 4.3. The condition number of C is equal to the largest singular value divided by the smallest one. Table 4.1,

index sinvals	sinvals matrix A	sinvals matrix B	sinvals matrix C	index sinvals	sinvals matrix A	sinvals matrix B	sinvals matrix C
0	70.207	52.678	14.096	22	0.967	0.000	0.349
1	37.385	23.130	7.258	23	0.828	0.000	0.286
2	34.691	21.188	6.628	24	0.812	0.000	0.138
3	29.137	19.545	6.264	25	0.801	0.000	0.118
4	26.238	18.122	5.647	26	0.764	0.000	0.116
5	24.426	16.956	5.170	27	0.736	0.000	0.114
6	21.866	15.989	4.848	28	0.733	0.000	0.110
7	20.371	15.209	4.485	29	0.722	0.000	0.105
8	18.632	14.637	4.204	30	0.688	0.000	0.105
9	17.089	14.249	3.935	31	0.506	0.000	0.105
10	16.092	14.033	3.693	32	0.224	0.000	0.100
11	11.131	2.769	3.560	33	0.100	0.000	0.072
12	5.687	0.000	1.590	34	0.095	0.000	0.000
13	5.023	0.000	0.820	35	0.095	0.000	0.000
14	4.519	0.000	0.735	36	0.084	0.000	0.000
15	4.153	0.000	0.661	37	0.077	0.000	0.000
16	3.763	0.000	0.605	38	0.077	0.000	0.000
17	3.666	0.000	0.541	39	0.071	0.000	0.000
18	3.473	0.000	0.529	40	0.071	0.000	0.000
19	3.465	0.000	0.500	41	0.065	0.000	0.000
20	3.348	0.000	0.499	42	0.032	0.000	0.000
21	2.006	0.000	0.481	43	0.000	0.000	0.000

Table 4.1: Table of the singular values of the matrices A, B and C

column 4 shows that matrix C is ill-conditioned. Matrix $C^T\ C$ is positive definite, thus its singular value decomposition is equal to its eigenvalue decomposition. In the example (Figure 4.3) twelve eigenvalues give good results, but 34 eigenvalues should give a slightly better result.

It is not necessary to use the formula $r = n - t - 1$ to compute the number of significant singular values for matrix B. A smaller number also is acceptable, because there is a last correction in the eigenvalue decomposition of $C^T\ C$. In Figure 4.1 (table 4.1, column 2) r may be twelve or twenty two.

Chapter 5

Model functions

In section 1.3 a velocity function u and a heigt function h have been defined. These model functions satisfied some conditions. When the conditions change, these model functions are not applicable. In this chapter a number of velocity functions with corresponding height functions are defined coping with various conditions.

5.1 Problem definition

Consider a 2-D cavity with a liquid film at the bottom, see Fig. 1.1. At rest the surface of the liquid is at distance h_0 from the bottom. The motion of the thin liquid film due to surface gradients with and without gravity is modelled. The dimensionless coordinates x and y and the dimensionfull coordinates x' and y' are related through $x' = L\ x$ and $y' = h\ y$, see also section 1.2. The flow domain may have a left and/or a right boundary and it might be anti-symmetric around $x' = 0$. Without conditions the flow domain is $-\infty < x' < \infty,\ 0 < y' < h$. The Marangoni flow is localized so that in this case it suffices to restrict the domain of interest to a domain $0 < x' < L,\ 0 < y' < h$, where L is chosen properly. In case of flows anti-symmetric around $x' = 0$ and with a boundary at $x = L$ the domain is $-L < x' < L,\ 0 < y' < h$. Because of the anti-symmetry it suffices to consider the domain $0 < x' < L,\ 0 < y' < h$.

5.2 Expansions

For the liquid velocity component in x-direction, $u(x, y)$, the following expansion is exployed:

$$u = \sum_{i=0}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} d_{ik}\ f(k,x) \left(\frac{y}{h(x)}\right)^{i+1}$$

Different model functions $f(k, x)$ are used, e.g.:
$f(0, x) = x$ and $f(k, x) = \sin(\pi\ k\ x)$ for $k > 0$
or

$f(k,x) = x^{k+1}$ for $k \geq 0$.

In these two examples the velocities $u(x,y)$ are zero for $x = 0$, but in the second example the derivates at $x = 0$ generally do not vanish. In the first example the $k = 0$ term makes it possible to model flows without a solid boundary at $x = 1$. Convenient choices for f without extra conditions like symmetry are

$$f(k,x) = (x-a)^k, \;\; k \geq 0$$

where a is a number between 0 and 1, e.g. $a = \frac{1}{2}$,

$$\text{and } f(k,x) = T_k(2x-1)$$

T_k, $k \geq 0$ are the Chebyshev polynomials. They are defined for $-1 \leq x \leq 1$ by $T_k(x) = \cos(k \text{ arccos } x)$. The first three are $T_0(x) = 1$, $T_1(x) = x$ and $T_2(x) = 2x^2 - 1$. For every $k \geq 2$, $T_k(x)$ can be deduced from the recursion relation $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$.

If the flow profile is anti-symmetric, i.e. $u(x,y,t) = -u(-x,y,t)$, we restrict ourselves to odd Chebyshev polynomials $f(k,x) = T_{2k+1}(x)$. All zeros of a Chebyshev polynomial and all its extremes are situated in the interval of interest, $-1 \leq x \leq 1$, and all extremes are equal to 1 or -1. This oscillation between extreme values of equal magnitude is known as the equal-ripple property. The equal-ripple property ensures that the fitting error is evenly spread over the entire interval.

5.3 The liquid height

In the model with $f(k,x) = \sin(\pi(k+1)x)$, $k \geq 0$ a solid boundary occurs at $x = 1$. For the liquid height we then choose the following function h

$$h(x) = \sum_{m=0}^{\texttt{Ncoef\_b}} b_m \cos(\pi m x)$$

This function is symmetric with regard to $x = 0$ and $\frac{\partial h}{\partial x}(1) = 0$. This last feature makes this expansion unsuitable for the model with $f(0,x) = x$, $f(k,x) = \sin(\pi k x)$, $k > 0$. In that case the following model expansion is used

$$h(x) = b_0 x^2 + \sum_{m=1}^{\texttt{Ncoef\_b}} b_m \cos(\pi(m-1)x)$$

For the models $f(k,x) = x^{k+1}$ and $f(k,x) = (x - \frac{1}{2})^k$, $k \geq 0$ it stands to reason to use

$$h(x) = \sum_{m=0}^{\texttt{Ncoef\_b}} b_m x^m$$

If $f(k,x) = T_k(2x-1)$, a good candidate for the height function is

$$h(x) = \sum_{m=0}^{\texttt{Ncoef\_b}} b_m T_k(2x-1)$$

If $f(k,x) = T_{2k+1}(x)$ the function h has to be symmetric in $x = 0$, so a sensible choice is

$$h(x) = \sum_{m=0}^{\texttt{Ncoef\_b}} b_m T_{2k}(x)$$

5.4 Time evolution of the height

To determine accurately the new parameters b_m in the expansion of h the coefficients of corresponding base functions on both sides of the equation $\frac{\partial h}{\partial t} = v - u\frac{\partial h}{\partial x}$ are set equal. However, this is not always possible. In that case the new values for the parameters b_m are defined by calculating $\frac{\partial h}{\partial t}$ and $\frac{\partial^2 h}{\partial x \partial t}$ for a number of fixed points at the interface. By integrating these expressions in time with the aid of a Adams-Bashford method the functions h and $\frac{\partial h}{\partial x}$ are computed for these points. After substitution in the expansion for h the b-values are calculated. In appendices D, E and F derivations are given of the expressions used for the determination of new values of the b-vector for a number of models.

Chapter 6

Towards a global velocity function with fewer parameters

Consider a two-dimensional liquid film set in motion by Marangoni stresses at the surface. There is an axis of symmetry at x=0; y=0 is at the bottom of the cavity, see Fig. 1.1. The x-direction is scaled such that at x=1 and x=-1 either a solid boundary occurs or there is no motion of the fluid. In most applications to be considered, the largest liquid velocities are found in the region $-\frac{1}{2} < x < \frac{1}{2}$ and $\frac{1}{2} < y \leq 1$.

A model of the horizontal velocity component u to be valid in the whole domain $-1 \leq x \leq 1 \quad 0 \leq y \leq \infty$ of the form

$$u(x,y,t) = \sum_{i=0}^{\mathtt{nic}-1} \left(\sum_{k=0}^{\mathtt{nkc}-2} d_{i,k}(t) \sin(\pi(k+1)x) + d_{i,\mathtt{nkc}-1} x \right) \left(\frac{y}{h(x)} \right)^{i+1} \tag{6.1}$$

has both positive and negative features. Here h denotes the the height function and $\{d_{i,k}\}$ is a set of nic.nkc time dependent variables. A positive feature is e.g. that the velocity is a linear combination of the unknowns $d_{i,k}$. When the velocity is known, the unknowns $d_{i,k}$ can easily be obtained by solving a matrix equation. A negative feature is that the number of unknowns is rather large. Another negative feature is that the model does not take into account that in some regions hardly any motion of the liquid occurs.

It is attempted in this study to find better model functions $u(x,y,t)$ for cases when fluid motion is mainly limited by a certain part of the film.

Let us ussume that the region $\frac{1}{2} < x < 1$ the velocities are relatively small. A damping factor is introduced to take this into account. So, instead of $\sin(\pi(k+1)x)$ we may write $e^{-a_k x} \sin(\pi(k+1)x)$ with $a_k > 0$. In the model 6.1 u is a sum of sines with periods which are determined beforehand. It might be better to transfer the determination of the periods to the algorithm itself. In some cases extra freedom might reduce the number of parameters to be determined. A convenient model function therefore contains factors of the form $e^{-a_k x} \sin(\alpha_k x)$.
To cope with the differences in velocities in the upper (i.e. $y > \frac{h}{2}$) and lower part of the liquid the damping factor $e^{d_k \frac{y}{h(x)}} - 1$ is selected with $d_k > 0$. Note that this factor is zero

at $y=0$ making $u(x,0,t)=0$, and has its maximimum value (e^d-1) at the interface. The factor $\left(\frac{y}{h(x)}\right)^{i+1}$ in the model of the preceding section is replaced by $\cos(\beta_k(\frac{y}{h(x)}-1))$. This factor is maximum at $y=h(x)$, i.e. at the interface. So far we have created the following velocity model

$$u(x,y,t)=\sum_{k=0}^{\mathtt{nkc}-1}\gamma_k e^{-a_k x}\sin(\alpha_k x)\left(e^{d_k\frac{y}{h(x)}}-1\right)\cos(\beta_k(\frac{y}{h(x)}-1))$$

The number of parameters in this expansion is 5nkc. The velocity u is not linear in the parameters any more. The equations obtained from the contour integrals, the time-differentiated tangential stress condition and the vorticity equation at the bottom are linear in the unknowns, which are the time derivatives $\frac{\partial\alpha_k}{\partial t}$, $\frac{\partial\beta_k}{\partial t}$, $\frac{\partial\gamma_k}{\partial t}$, $\frac{\partial a_k}{\partial t}$, $\frac{\partial d_k}{\partial t}$. Using the Adams Bashford formula, the parameters for u at the next instance of time are computed. The only problem is the tangential stress condition at time zero. Since at the start of the program the tangential stress condition is not linear in the unknowns. So this problem is omitted in the program. Owing to this the user has to provide appropriate starting values.

Another drawback of this expansion of u is that the formula derived from u are too complex to be calculated by hand. The program `Maple`$^{\mathrm{TM}}$, which can manipulate mathematical formulae, has been used to perform the necessary derivations and reductions.

The vertical velocity component at the bottom (y=0), v, is derived with the continuity equation and reads

$$v(x,0,t)=\sum_{k=0}^{\mathtt{nkc}-1}\frac{\gamma_k d_k}{\beta_k(\beta_k^2+d_k^2)}e^{-a_k x}(\beta_k\cos(\beta_k)+d_k\sin(\beta_k))$$

$$(a_k h(x)\sin(\alpha_k x)-\alpha_k\cos(\alpha_k x)h(x)-\frac{\partial h}{\partial x}(x)\sin(\alpha_k x))$$

Due to the no-slip condition this expression should vanish at the bottom. This is the case if $d_k=-\beta_k\cot(\beta_k)$. This strong requirement is not acceptable, because it would reduce the number of independent parameters by 1.

To improve the above expression for u, the following expression is introduced, dependant on the positive integer n.

$$u_n(x,y,t)=\sum_{k=0}^{\mathtt{nkc}-1}\gamma_k e^{-a_k x}\sin(\alpha_k x)\left(\frac{y}{h(x)}\right)^n\cos(\beta_k(\frac{y}{h(x)}-1))$$

The corresponding vertical velocity component v at the bottom $(y=0)$ reads

$$\begin{aligned}v_n(x,y,t) &= \sum_{k=0}^{\mathtt{nkc}-1}\gamma_k(-1)^n\frac{d_k^n\sin(\beta_k)}{d\beta_k^n}\frac{n!}{\beta_k^{n+1}}\cdot\\ &\quad\cdot e^{-a_k x}(\alpha_k h(x)\cos(\alpha_k x)+\frac{\partial h}{\partial x}(x,t)\sin(\alpha_k x)-a_k h(x)\sin(\alpha_k x))\end{aligned}$$

Suitable parameters, e_n, have now to be found for which $\sum_{n=1}^{\infty} e_n v_n = 0$ for all x. There are infinitely many ways to accomplish this. One of the possibilities is the following. Choose if n is odd

$$e_n = -\frac{\tan(\beta_0)}{\beta_0}\frac{d_0^{n+1}}{n!}$$

And if n is even

$$e_n = \frac{d_0^n}{n!}$$

Then

$$\sum_{n=1}^{\infty} e_n v_n = \sum_{n=1}^{\infty}(e_{2n-1}v_{2n-1} + e_{2n}v_{2n}) = 0$$

because $e_{2n-1}v_{2n-1} + e_{2n}v_{2n} = 0$ for all $n > 0$. Conclusion: $v(x,0,t) = 0$ if

$$u(x,y,t) = \sum_{k=0}^{\mathrm{nkc}-1} \gamma_k e^{-\alpha_k x}\sin(\alpha_k x)\left(-\frac{\tan(\beta_k)}{\beta_k}d_k\cosh(\frac{d_k\,y}{h(x)}) + \sinh(\frac{d_k\,y}{h(x)}) - 1\right)\cos(\beta_k(\frac{y}{h(x)} - 1))$$

i.e. if

$$u(x,y,t) = \sum_{k=0}^{\mathrm{nkc}-1} \gamma_k e^{-\alpha_k x}\sin(\alpha_k x)\left(\left(1 - \frac{\tan(\beta_k)}{\beta_k}d_k\right)e^{\frac{d_k\,y}{h(x)}} + \left(1 + \frac{\tan(\beta_k)}{\beta_k}d_k\right)e^{-\frac{d_k\,y}{h(x)}} - 2\right)\cos(\beta_k(\frac{y}{h(x)} - 1))$$

Chapter 7

Tests and results

Consider the physical model described in section 1.2. The aspect ratio is $A = \frac{h_0}{L} = \frac{1}{80}$. The tank is filled with water with viscosity $\mu = 1.002 \cdot 10^{-3}$ $Pa\ s$ and mass density $\rho = 998\ \frac{kg}{m^3}$. The gravitational force is absent. Ethanol is added to the water surface at $x = 0$. By doing so the surface tension at $x = 0$ becomes $\sigma = \sigma_{ethanol} = 22.8 \cdot 10^{-3}\ \frac{N}{m}$ while at $x = 1$, $\sigma = \sigma_{water} = 72.3 \cdot 10^{-3}\ \frac{N}{m}$. This causes a surface tension gradient S_1. Figure 7.1 shows the assumed time history of the surface tension gradient profiles, S_1.

For $0 < t < \frac{1}{2}\sqrt{(2)}$, S_1 and S_{gr} are defined by:

$$S_1 = 4Lt(t-\beta)^2 \sin(\pi(1-(1-x)^{\frac{\beta}{t}}))\pi(1-x)^{\frac{\beta}{t}-1}\beta^{-3}(1+A^2\left(\frac{\partial h}{\partial x}\right)^2)^{-\frac{1}{2}} \tag{7.1}$$

and

$$S_{gr} = \frac{\sigma_{water}}{\Delta\sigma} - 4t^2(t-\beta)^2\beta^{-4}(1+\cos(\pi(1-(1-x)^{\frac{\beta}{t}}))) \tag{7.2}$$

with $\beta = \frac{\sqrt{2}}{2}$ and $\Delta\sigma = \sigma_{water} - \sigma_{ethanol}$.

In the examples the dimensionless timestep is 0.01, which corresponds to $0.108 \cdot 10^{-3}$ s. The number of fitting points is $ni \times nk = 50 \times 50$.

Figures 7.2 and 7.3 show the flow at time 0.2 s for x between 0 and 1. The velocity component in x-direction $u(x, y)$ is:

$$u(x,y) = \sum_{i=0}^{2}\sum_{k=1}^{10} d_{ik} f(k,x)\left(\frac{y}{h(x)}\right)\exp(\lambda_i(\frac{y}{h(x)}-1)) + \sum_{i=0}^{6}\sum_{k=1}^{10} d_{i+3,k} f(k,x)\left(\frac{y}{h(x)}\right)^{i+1} \tag{7.3}$$

with λ_0, λ_1 and λ_2 nonnegative constants and

$$f(k,x) = \sin(\pi(k+1)x) \tag{7.4}$$

and

$$h(x) = \sum_{m=0}^{12} b_m \cos(\pi m x) \tag{7.5}$$

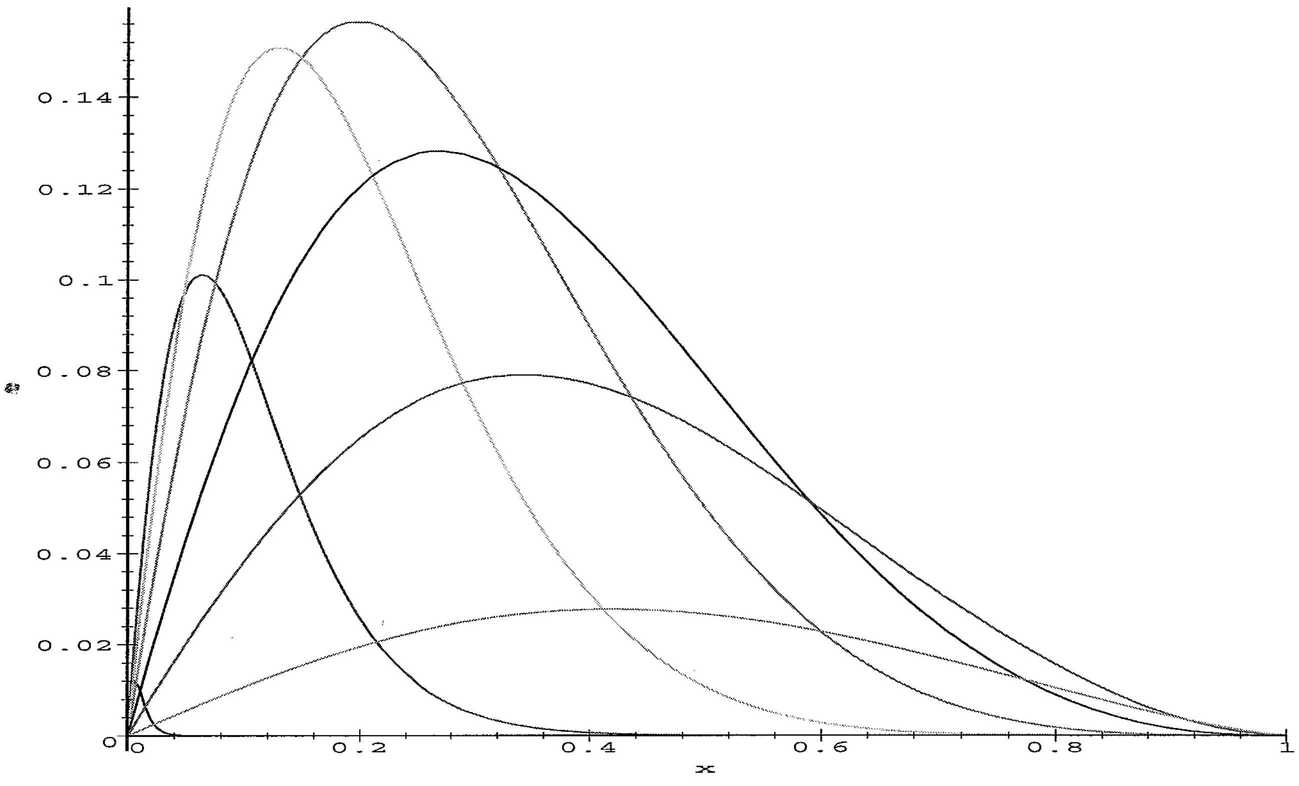


Figure 7.1: Assumed profiles of surface tension gradient S_1 for the instances $t = 0.01$, $t = 0.1$, $t = 0.2$, $t = 0.3$, $t = 0.4$, $t = 0.5$ and $t = 0.6$. The maximum of the profile of S_1 moves to the right in time.

If λ_0, λ_1 and λ_2 vanish, the formula for the velocity component 7.3 matches the expansion in section 5.2. If the water of the tank is at rest and the ethanol is added to the surface of the liquid then positive velocities are restricted to a very small strip near the interface. Fitting this velocity field with the expansion of section 5.2 results in a large signal to noise ratio (SNR). Therefore the expansion 7.3 is used with constants $\lambda_0 > 0$, $\lambda_1 \geq 0$ and $\lambda_2 \geq 0$. If the liquid water is replaced by a more viscious liquid, the penetration depth is larger and the expansion of section 5.2 suffice.

This model is symmetric with regard to $x = 0$. There is a solid boundary at $x = 1$.

Figures 7.2 and 7.3 show the flow at the instance $t = 0.1$ and $t = 0.2$. In Figure 7.2 $\lambda_0 = 100$, $\lambda_1 = 10$ and $\lambda_2 = 3$. In Figure 7.3 $\lambda_0 = 100$, $\lambda_1 = 62$ and $\lambda_2 = 34$. Notice the very slight influence of the chosen constants λ_1 and λ_2.

Figures 7.4 through 7.7 show the flow at the instances $t = 0.1$ and $t = 0.2$ for various models described in chapter 5 and the appendices, i.e. for various choises of the function f in 7.3. The parameters used are $ni = 50$, $nk = 50$, $nic = 10$, $nkc = 10$, $Ncoef_b = 12$, $\lambda_0 = 100$, $\lambda_1 = 10$ and $\lambda_2 = 3$.

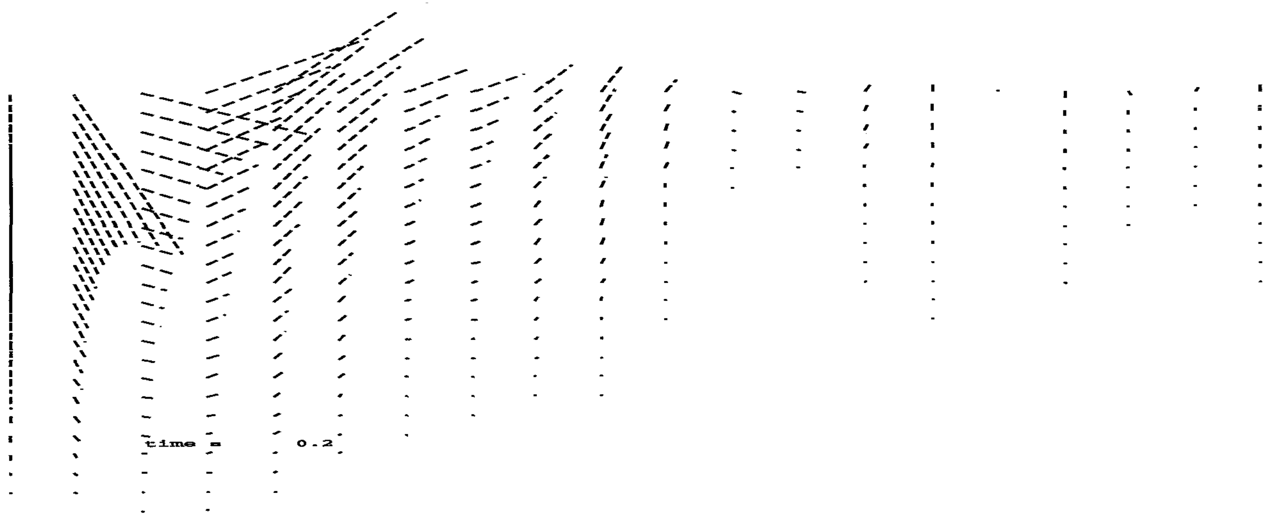


Figure 7.2: Flow caused by Marangoni convection at $t = 0.2$. In the model function 7.3 $f(x, y) = \sin(\pi(k+1)x)$, $\lambda_0 = 100$, $\lambda_1 = 10$ and $\lambda_2 = 3$.

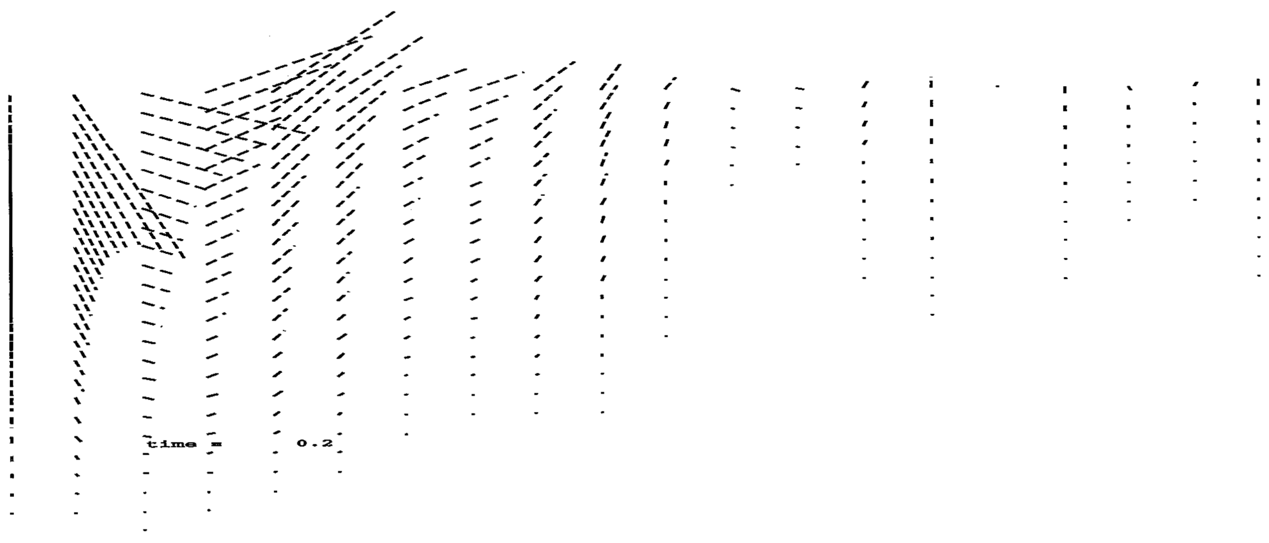
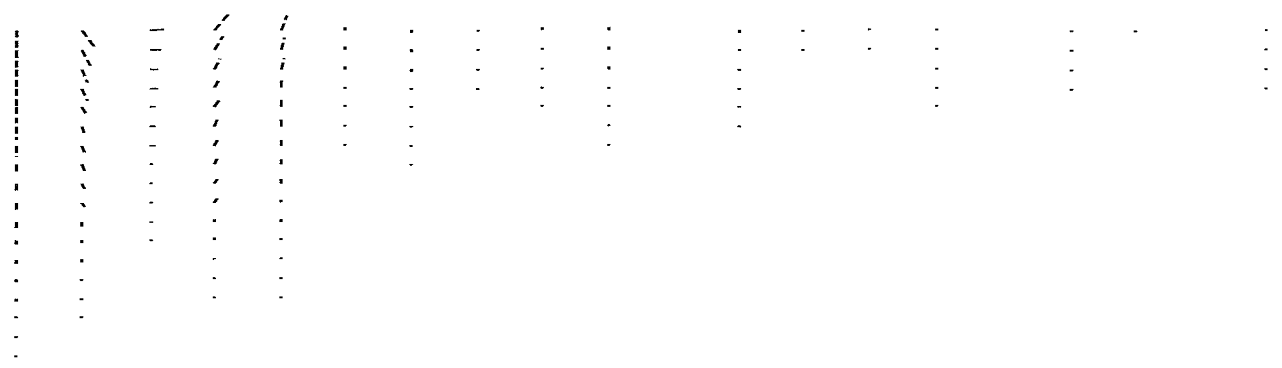


Figure 7.3: Flow caused by Marangoni convection at $t = 0.20$. In the model function 7.3 $f(x,y) = \sin(\pi(k+1)x)$, $\lambda_0 = 100$, $\lambda_1 = 62$ and $\lambda_2 = 34$.



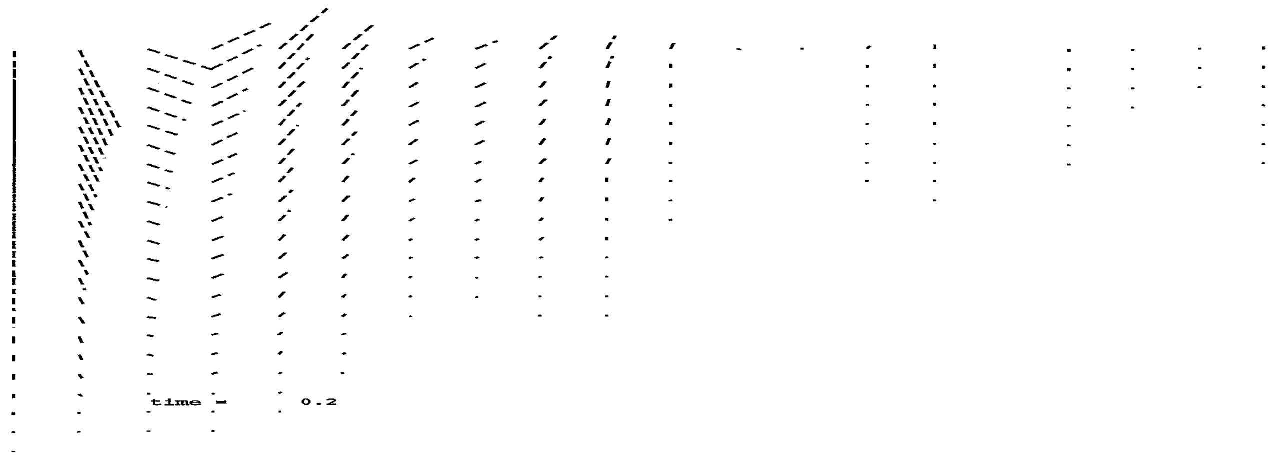
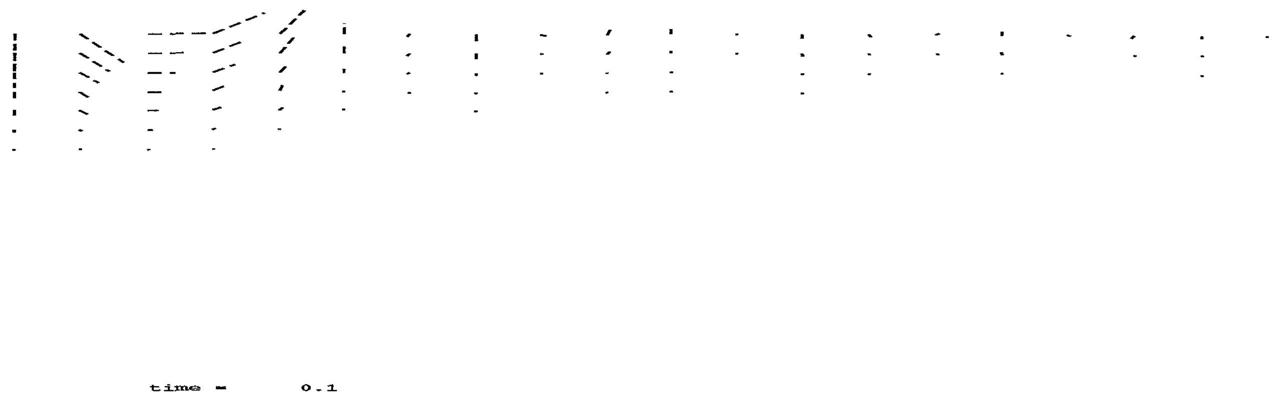


Figure 7.4: Flow caused by Marangoni convection. The function f used in the velocity model function is described in section 1.3. The figures show the velocity field at the instances $t = 0.1$ and $t = 0.2$



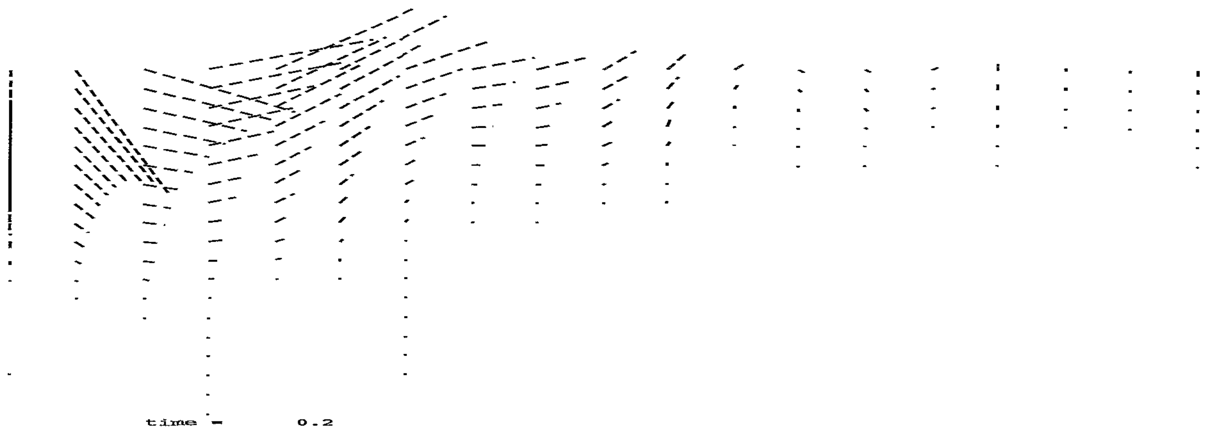
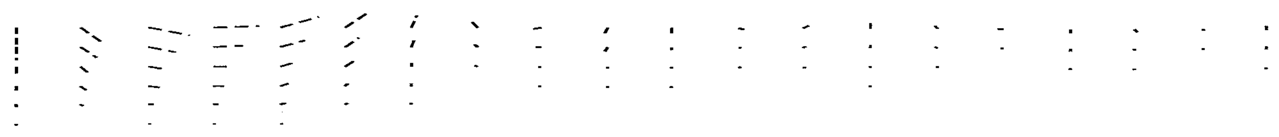


Figure 7.5: Flow caused by Marangoni convection. The function f used in the velocity model function is described in appendix E. The figures show the velocity field at the instances $t = 0.1$ and $t = 0.2$



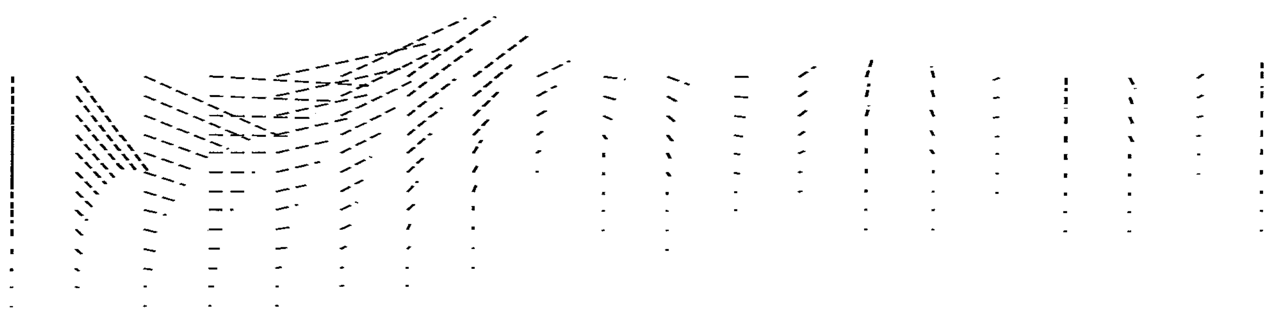
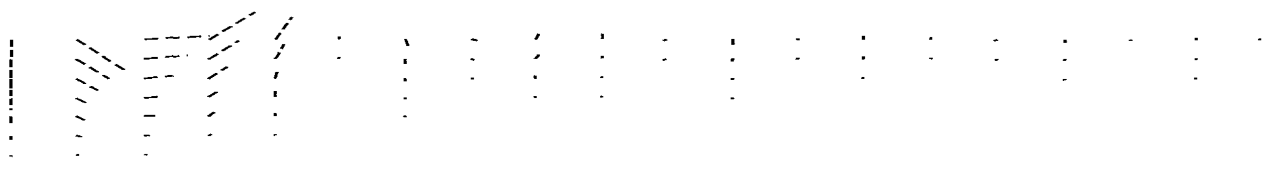


Figure 7.6: Flow caused by Marangoni convection. The function f used in the velocity model function is described in appendix F. The figures show the velocity field at the instances $t = 0.1$ and $t = 0.2$



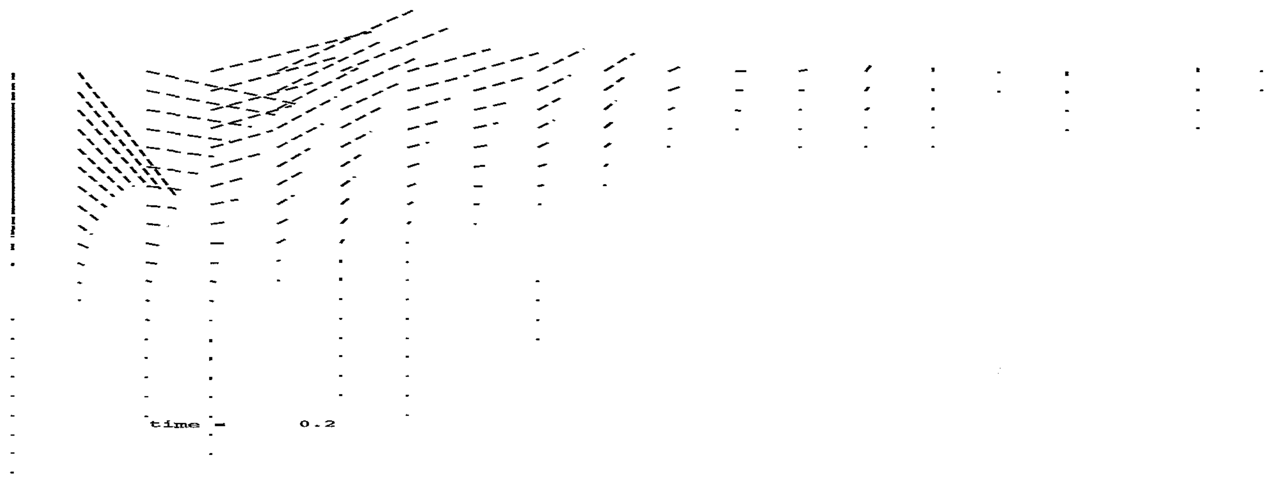


Figure 7.7: Flow caused by Marangoni convection. The function f used in the velocity model function is described in appendix D. The figures show the velocity field at the instances $t = 0.1$ and $t = 0.2$

7.1 Conclusions

In this report the time evolution of the velocity field and the corresponding interfacial behaviour is studied for a 2D cross-section of a rectangular cavity with a liquid layer at the bottom. Stress gradients at the surface of a liquid layer sets the fluid in motion. An algorithm is discussed which is flexible and modular in design which can cope with various configurations and geometries: The selection of a particular model for the horizontal velocity component depends on the features of the flow, such as where is the current strongest, and on conditions such as symmetry or boundary conditions. In chapter 5 many different models for the horizontal velocity component are shown. See also the appendices 4,5,6 and 7. In chapter 3 it was shown that the a right choice of integration paths is important for a good performance of the algorithm and depends on the features of the flow. In chapter 5 is shown a modelfunction for the horizontal velocity component for a cavity of infinite length, which rather deviates from the rest of the modelfunctions. Although the modelfunction is not linear in the parameters, the algorithm is fast and needs only a few parameters.

Bibliography

[1] BATCHELOR, GK - *An Introduction to Fluid Dynamics*, Cambridge University Press, (1970).

[2] VAN DEN BOS, A, - *Handbook of Measurement Science*, J. Wiley & Sons Ltd, (1982)).

[3] LAWSON, CL & HANSON, RJ, -*Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, N.J. (1974).

[4] VAN DE GELD, CWM & BROUWERS, HWH - *The condensate heat resistance for dropwise condensation in plastic compact heat exchangers*, Accepted for publication in Wärme- und Stoffübertragung, (1995).

[5] MARRA, J & HUETHORST, JAM - *Physical principles of Marangoni drying*, Langmuir, (1991), 7.

[6] SCHWABE, D,DUPONT, O,QUEEKS, P & LEGROS, JC - *Experiments on Marangoni-Bénard instability problems under normal and microgravity conditions. Proceedings 7th European Symposium on Materials and Fluid Sciences in Microgravity*, Oxford, UK, (1989), 7, ESA SP-295 (1990).

[7] ZHANG, NENGLI & YANG, WEN-JEI, - *Evaporative convection in minute drops on a plate with temperature gradient*, Int. J. Heat Mass Transfer, vol. 26, no 10, pp 1479-1488, (1983).

[8] OSTRACH, S & PRADHAN, A. - *Surface-tension induced convection at reduced gravity*, AAIA Journal, vol. 16, no.5, (1978), 5.

[9] Hoefsloot, HCJ, - *Marangoni convection under microgravity conditions*, Ph.D. Thesis, Groningen University, The Netherlands, (1992).

[10] KAO, YS & KENNING, DBR, - *Thermocapillary flow near a hemispherical bubble on a heated wall*, J. Fluid. Mech., vol 53, part 4, pp. 715-735, (1972).

[11] LEGROS, JC, LIMBOURG-FONTAINE, MC & PETRE, G, - *Surface tension minimum and Marangoni convection*, Proceedings of symposium: Fluid dynamics and space, Rhode-Saint-Genèse, ESA SP-265 (1986).

[12] WOZNIAK, K & WOZNIAK, G, - *Particle-image velocimetry applied to thermocapillary convection*, Exp. in fluids 10, 12-16 (1990).

[13] WOZNIAK, K, WOZNIAK G & RÖSGEN, T, - *Particle-image-velocimetry applied to thermocapillary convection.* Exp. in Fluids 10 12-16, (1990).

Appendix A

Derivation of the tangential stress condition

At every point of the interface the surface stress is decomposed into a stress vector parallel to the surface (called the tangential stress vector) and a stress vector perpendicular to the surface (called the normal stress vector). The tangential stress condition is derived as follows.

Consider a tangential vector t of length one and its corresponding normal vector n, also of length one. Define

$$N = \sqrt{1 + \left(\frac{\partial h}{\partial x}\right)^2}$$

Then

$$t = \begin{pmatrix} \frac{1}{N} \\ \\ \frac{1}{N}\frac{\partial h}{\partial x} \end{pmatrix} \text{ and } n = \begin{pmatrix} -\frac{1}{N}\frac{\partial h}{\partial x} \\ \\ \frac{1}{N} \end{pmatrix}$$

For the tangential stress condition we use the relation [1]:

$$t^T \tau_{fluid} n + t^T \tau_{air} n = D[\sigma]\, t \tag{A.1}$$

with

$$D[\sigma] = \begin{pmatrix} \frac{\partial \sigma}{\partial x} & \frac{\partial \sigma}{\partial y} \end{pmatrix}$$

$$\tau_{fluid} = \begin{pmatrix} -p_{fluid} + 2\mu_{fluid}\frac{\partial u}{\partial x} & \mu_{fluid}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \\ \\ \mu_{fluid}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) & -p_{fluid} - 2\mu_{fluid}\frac{\partial u}{\partial x} \end{pmatrix}$$

By changing the index $fluid$ in τ_{fluid} into air we get the matrix τ_{air}. The dynamic viscosity μ_{air} of the gas is much less than μ_{liquid}, so that in A.1 τ_{air} can be replaced by $-p_{air}I$, with

I the identity matrix. Substitution yields

$$\frac{\mu_{fluid}}{N^2}\left\{\left(\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}\right)\left(2-N^2\right)-4\frac{\partial h}{\partial x}\frac{\partial u}{\partial x}\right\}=\frac{1}{N}\frac{\partial \sigma}{\partial x}+\frac{1}{N}\frac{\partial h}{\partial x}\frac{\partial \sigma}{\partial y}$$

Derivation of the normal stress condition

For the normal stress condition we use the relation [1]:

$$n^T\tau_{fluid}n+n^T\tau_{air}n=-\sigma(\frac{1}{R_1}+\frac{1}{R_2})$$

R_1 and R_2 are the radii of curvature at the point of consideration. $R_2=0$, because there is no third dimension.

$$\frac{1}{R_1}=\frac{1}{N^3}\frac{\partial^2 h}{\partial x^2}$$

Substitution yields

$$\left(p_{fluid}-p_{air}-2\mu_{fluid}\frac{\partial u}{\partial x}\right)\left(\frac{\partial h}{\partial x}\right)^2-2\mu\left(\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}\right)\frac{\partial h}{\partial x}+p_{fluid}-p_{air}-2\mu_{fluid}\frac{\partial v}{\partial y}$$

$$=-\sigma\frac{1}{N}\frac{\partial^2 h}{\partial x^2}$$

From the formula derived for the tangential stress condition we know

$$\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}=\frac{1}{2-N^2}\left\{\frac{N}{\mu_{fluid}}\frac{\partial \sigma}{\partial x}+\frac{N}{\mu_{fluid}}\frac{\partial h}{\partial x}\frac{\partial \sigma}{\partial y}+4\frac{\partial h}{\partial x}\frac{\partial u}{\partial x}\right\}$$

Substitution into the normal stress condition yields after some algebraic manipulation

$$p_{fluid}-p_{air}=-\sigma\frac{1}{N^3}\frac{\partial^2 h}{\partial x^2}-\frac{2N^2}{2-N^2}\frac{\partial u}{\partial x}-\frac{2}{N(2-N^2)}\frac{\partial h}{\partial x}\left(\frac{\partial \sigma}{\partial x}+\frac{\partial h}{\partial x}\frac{\partial \sigma}{\partial y}\right)$$

Appendix B

The kinematic boundary condition

There are several ways to derive the formula $\frac{\partial h}{\partial t} = v - u\frac{\partial h}{\partial x}$. Here we give two.

First derivation:

Consider the part of the fluid between $x = x_0$ and $x = x_0 + \Delta x$. The amount of fluid passing the line $x = x_0$ during Δt seconds is

$$\int_0^{h(x_0)} u(x_0, y, t)\ \Delta t\ dy$$

and the amount passing the line $x = x_0 + \Delta x$ is

$$\int_0^{h(x_0+\Delta x)} u(x_0 + \Delta x, y, t)\ \Delta t\ dy$$

The increase of the amount of fluid between $x = x_0$ and $x = x_0 + \Delta x$ is given by

$$\int_{x_0}^{x_0+\Delta x} (h(x, t + \Delta t) - h(x, t))dx$$

$$= \int_0^{h(x_0,t)} u(x_0, y, t)\Delta t\ dy - \int_0^{h(x_0+\Delta x,t)} u(x_0 + \Delta x, y, t)\Delta t\ dy$$

Substitute

$$u(x_0 + \Delta x, y, t) = u(x_0, y, t) + \Delta x\ \frac{\partial u}{\partial x}(x_0, y, t) \text{ and}$$

$$h(x_0 + \Delta x, t) = h(x_0, t) + \Delta x\ \frac{\partial h}{\partial x}(x_0, t)$$

and divide both sides by $\Delta x\ \Delta t$ and use the general property

$$\lim_{\Delta a\to 0} \int_a^{a+\Delta a} f(p)dp = f(a)$$

This yields

$$v(x_0, h(x_0, t) - u(x_0, h(x_0), t)\frac{\partial h}{\partial x}(x_0, t) = \frac{\partial h}{\partial t}(x_0, t)$$

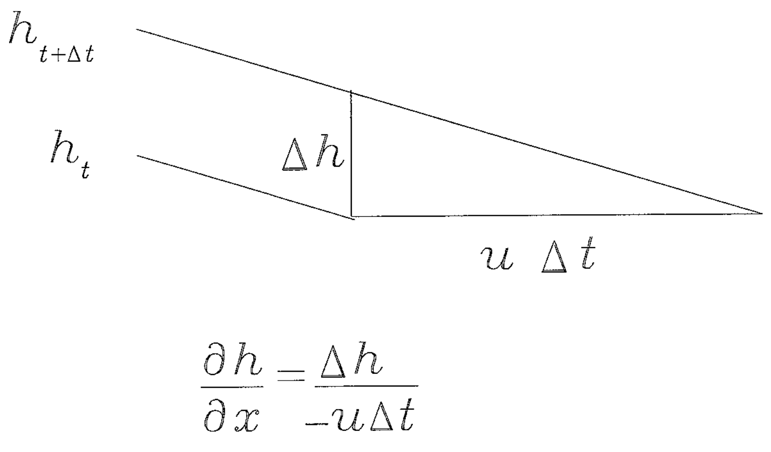

$h_{t+\Delta t}$
h_t
Δh
$u\ \Delta t$
$\frac{\partial h}{\partial x} = \frac{\Delta h}{-u\Delta t}$

Figure B.1: The kinematic boundary condition in the case that there are no vertical velocity components

Second Derivation:

Suppose there are only vertical velocities. Then $\frac{\partial h}{\partial t} = v$. On the other hand, if there are only horizontal velocities (see Figure B.1)

$$\frac{\partial h}{\partial t} = -u\frac{\partial h}{\partial x}$$

Thus, in general

$$\frac{\partial h}{\partial t} = v - u\frac{\partial h}{\partial x}$$

Appendix C

Transformation of coordinates

The first order Taylor expansion for the pressure p in (x_0, y_0) is

$$p(x,y) = p(x_0,y_0) + \begin{pmatrix} \frac{\partial p}{\partial x}(x_0,y_0) & \frac{\partial p}{\partial y}(x_0,y_0) \end{pmatrix} \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix}$$

Substitution of $x = x_0 + \Delta x, y = c\,h(x_0)$ and $y = c\,h(x_0 + \Delta x)$ yields

$$p(x_0 + \Delta x, c\,h(x_0 + \Delta x)) - p(x_0, c\,h(x_0)) = \begin{pmatrix} \frac{\partial p}{\partial x} & \frac{\partial p}{\partial y} \end{pmatrix} \begin{pmatrix} \Delta x \\ c(h(x_0 + \Delta x) - h(x_0)) \end{pmatrix}$$

Choose a value between 0 and 1 for c. If s is the coordinate along the path described by the equation $y = c\,h(x)$ then

$$\Delta p = \begin{pmatrix} \frac{\partial p}{\partial x} & \frac{\partial p}{\partial y} \end{pmatrix} \begin{pmatrix} 1 \\ c\,\frac{\partial h}{\partial x} \end{pmatrix} \Delta x$$

and thus

$$\int_{(x_1, c\,h(x_1))}^{(x_2, c\,h(x_2))} \frac{\partial p}{\partial s} ds = \int_{x_1}^{x_2} \left(\frac{\partial p}{\partial x} + c\,\frac{\partial h}{\partial x}\frac{\partial p}{\partial y} \right) dx$$

Appendix D

A model for the horizontal velocity component satifisfying the no-slip condition at three boundaries

Suppose a solid boundary occurs at $x = 1$ and the velocity component u is odd in x, i.e. $u(x, y, t) = -u(-x, y, t)$. The problem is to find a function f, see section 5.2 such, that the velocity component u satisfies the conditions. The conditions for the velocity component u are translated into conditions for the function f. The function f has to satisfy the following conditions for $l \geq 0$: 1. f is odd in x, i.e. $f(l, -x) = -f(l, x)$. 2. $f(l, 1) = 0$. 3. $\frac{\partial f}{\partial x}(l, 1) = 0$.

To prove condition 1, notice that $u(x, y, t) = -u(-x, y, t)$ for all possible choices of the coefficients d_{ik} in the expansion of u. Substitute $d_{0l} = 1$ and $d_{ik} = 0$ for $i \neq 0$ or $k \neq l$ into the equation $u(x, y, t) = -u(-x, y, t)$ and notice that $h(-x) = h(x)$.

Conditions 2 and 3 are consequences of the no-slip conditions $u(1, y, t) = 0$ and $v(1, y, t) = 0$. Substitution of $d_{0l} = 1$ and $d_{ik} = 0$ for $i \neq 0$ or $k \neq l$ into the equations $u(1, y, t) = 0$ and $v(1, y, t) = 0$ yields respectively $f(l, 1) = 0$ and $\frac{\partial f}{\partial x}(l, 1) = 0$.

The reverse is also true: if the function f satisfies the abovementioned three conditions, then u is odd in x and u satisfies the no-slip condition at $x = 1$. For the construction of a function f that satisfies the three boundary conditions, the solutions of the following characteristic value problem are used

$$\frac{d^4 y}{dx^4} = \alpha^4 y.$$

The boundary conditions are

$$y(-1) = y(1) = \frac{dy}{dx}(-1) = \frac{dy}{dx}(1) = 0$$

and y is odd. Let α_k denote a characteristic value and let y_k be the proper solution belonging to it. Then, multiplying the equation governing y_k by y_n (belonging to α_n) and integrating over the range of x, we obtain

$$\alpha_k^4 \int_{-1}^{1} y_k y_n dx = \int_{-1}^{1} y_n \frac{d^4 y_k}{dx^4} dx$$

Transforming the integral on the right-hand side by two successive integrations by parts, we are left with

$$\alpha_k^4 \int_{-1}^{1} \frac{d^2 y_n}{dx^2} \frac{d^2 y_k}{dx^2} dx,$$

the integrated parts vanishing (both times) on account of the boundary conditions on y_n. From the symmetry of the integral on the right-hand side of the last equation in n and k, it follows that

$$(\alpha_k^4 - \alpha_n^4) \int_{-1}^{1} y_k y_n dx = 0$$

Hence, for $k \neq n$

$$\int_{-1}^{1} y_k y_n dx = 0$$

The solutions y_k, therefore, form an orthogonal set. It can be shown that this set is complete, i.e. every continuous function in the interval $(-1, 1)$ which together with its first derivative vanishes at the endponts of the interval is a infinite linear combination of the elements of the set $\{y_n, n > 0\}$. The proper solutions of the fourth order differential equation and its boundary conditions are readily found.

$$y_k = f(x, k) = \frac{\sinh(\lambda_k x)}{\sinh(\lambda_k)} - \frac{\sin(\lambda_k x)}{\sin(\lambda_k)}$$

Defined in this manner, the function f clearly vanish at $x = 1$; that the derivative in x also vanish at this point requires that λ_k is the root of the characteristic equation

$$\coth(\lambda) + \cot(\lambda) = 0.$$

It can be readily verified that function $f(x, k)$ satisfies

$$\int_{-1}^{1} f(x, k) f(x, n) dx = \delta_{kn}.$$

For $k > 4$ the asymptotic formula

$$\mu_k \sim (2k + \frac{1}{2}) \frac{\pi}{2}$$

gives the roots correct to ten decimals. Further, $\mu_1 = 3.92660231, \mu_2 = 7.06858274, \mu_3 = 10.21017612, \mu_4 = 13.35176878$.

Now suppose that a solid boundary occurs at $x = 0$ and $x = 1$. The problem is to find a function f such, that the velocity component u satisfies the no-slip conditions. The conditions for the function f now read for $l \geq 0$: 2. $f(l, 0) = f(l, 1) = 0$. 3. $\frac{\partial f}{\partial x}(l, 0) = 0,\ frac\partial f \partial x(l, 1) = 0$.

Consider the characteristic value problem defined by the equation

$$\frac{d^4y}{dx^4} = \alpha^4 y.$$

and the boundary conditions

$$y(0) = y(1) = \frac{dy}{dx}(0) = \frac{dy}{dx}(1) = 0$$

The proper solutions of these equations fall into two non-combining groups

$$y_k = f_1(x,k) = \frac{\cosh(\lambda_k(2x-1))}{\cosh(\lambda_k)} - \frac{\cos(\lambda_k(2x-1))}{\cos(\lambda_k)}$$

and

$$y_k = f_2(x,k) = \frac{\sinh(\mu_k(2x-1))}{\sinh(\mu_k)} - \frac{\sin(\mu_k(2x-1))}{\sin(\mu_k)}$$

These functions clearly vanish at $x = 0$ and $x = 1$. Their derivatives also vanish at these points if

$$\tanh(\lambda) + \tan(\lambda) = 0$$

and

$$\coth(\mu) - \cot(\mu) = 0$$

The functions f_1 and f_2 have the property

$$\int_0^1 f_i(x,k) f_j(x,n)dx = \delta_{ij}\delta_{k,n}$$

For $k > 4$ the asymptotic formulae

$$\lambda_k \sim (2k - \frac{1}{2})\frac{\pi}{2}$$

and

$$\mu_k \sim (2k + \frac{1}{2})\frac{\pi}{2}$$

give the roots correct to ten decimals. Further, $\lambda_1 = 2.36502037$, $\lambda_2 = 5.49780392$, $\lambda_3 = 8.63937983$, $\lambda_4 = 11.78097245$ and $\mu_1 = 3.92660231$, $\mu_2 = 7.06858274$, $\mu_3 = 10.21017612$, $\mu_4 = 13.35176878$.

Appendix E

The treatment of the kinematic boundary condition for the case of sine expansion with one linear term

The expansion of the fluid velocity in x-direction is the following linear combination of sines plus an extra linear term in x

$$u(x,y,t) = \sum_{i=0}^{\texttt{nic}-1} \left\{ \sum_{k=0}^{\texttt{nkc}-2} d_{ik}(t)\sin(\pi(k+1)x) + d_{i,\texttt{nkc}-1}x \right\} \left(\frac{y}{h(x,t)}\right)^{i+1}$$

The corresponding formula for the height looks awkward

$$h(x,t) = \sum_{l=0}^{\infty}\sum_{m=0}^{\infty} \tau_{l,m} x^l si(l,m)$$

Here the function si is defined as follows:
if l is even, then $si(l,m) = \cos(\pi m x)$
if l is odd, then $si(l,m) = \sin(\pi m x)$

Only a finite number of parameters $\tau_{l,m}$ are unequal to zero. This number increases in time. Let the initial condition be given by $t = 0$ and $\tau_{0,0} = 1$ and for $l+m > 0$ is $\tau_{l,m} = 0$. The definition of si yields

$$\frac{\partial si}{\partial x}(l,m) = (-1)^l \pi m \; si(l-1,m)$$

Define for $0 \leq k < \texttt{nkc}$, $\alpha_k = \sum_{i=0}^{\texttt{nic}-1} \frac{d_{ik}}{i+2}$. Then, after some algebraic manipulations

$$v - u\frac{\partial h}{\partial x} \quad = \quad -\sum_{k=1}^{\texttt{nkc}-1} \alpha_{k-1}\frac{\partial}{\partial x}(h\sin(\pi k x)) - \alpha_{\texttt{nkc}-1}\frac{\partial}{\partial x}(h\ x)$$

$$
\begin{aligned}
= \quad & -\frac{\partial}{\partial x}\left(\sum_{l=0}^{\infty}\sum_{m=0}^{\infty}\sum_{k=1}^{\mathrm{nkc}-1}\alpha_{k-1}\tau_{l,m}x^{l}\frac{(-1)^{l}}{2}(si(l-1,m+k)-si(l-1,m-k))\right)\\
& -\frac{\partial}{\partial x}\left(\alpha_{\mathrm{nkc}-1}\sum_{l=0}^{\infty}\sum_{m=0}^{\infty}\tau_{l,m}x^{l+1}si(l,m)\right)\\
& =\sum_{l=0}^{\infty}\sum_{m=0}^{\infty}\sum_{k=1}^{\mathrm{nkc}-1}\alpha_{k-1}\tau_{l,m}x^{l-1}\frac{(-1)^{l+1}}{2}(si(l-1,m+k)-si(l-1,m-k))\\
& +\sum_{l=0}^{\infty}\sum_{m=0}^{\infty}\sum_{k=1}^{\mathrm{nkc}-1}\alpha_{k-1}\tau_{l,m}x^{l}\frac{\pi}{2}(-(m+k)si(l,m+k)+(m-k)si(l,m-k))\\
& -\alpha_{\mathrm{nkc}-1}\sum_{l=0}^{\infty}\sum_{m=0}^{\infty}\tau_{l,m}(l+1)x^{l}si(l,m)\\
& +\alpha_{\mathrm{nkc}-1}\sum_{l=0}^{\infty}\sum_{m=0}^{\infty}\tau_{l,m}\pi m(-1)^{l}x^{l+1}si(l-1,m)
\end{aligned}
$$

Comparing coefficients of si with the same arguments on both sides of this equation yields the following formula for the time-derivative of $\tau_{l,r}$

$$
\begin{aligned}
\frac{\partial\tau_{l,r}}{\partial t} \quad = \quad & -\sum_{l=0}^{\infty}\left[\sum_{r=1}^{\mathrm{nkc}-1}\left\{\sum_{r=1}^{k}\frac{\alpha_k}{2}\left((-1)^{l+1}(l+1)\tau_{l+1,r-k}+\pi r\tau_{l,r-k}\right)\right.\right.\\
& \left.+\sum_{k=r}^{\mathrm{nkc}-1}\frac{\alpha_{k-1}}{2}\left((l+1)\tau_{l+1,k-r}+(-1)^{l}\pi r\tau_{l,k-r}\right)\right\}\\
& +\sum_{r=\mathrm{nkc}}^{\infty}\sum_{k=1}\mathrm{nkc}-1\frac{\alpha_{k-1}}{2}\left((-1)^{l+1}(l+1)\tau_{l+1,r-k}+\pi r\tau_{l,r-k}\right)\\
& +\sum_{r=0}^{\infty}\sum_{k=1}\mathrm{nkc}-1\left\{\frac{\alpha_{k-1}}{2}\left((-1)^{l}(l+1)\tau_{l+1,r+k}+\pi r\tau_{l,r+k}\right)\right.\\
& \left.\left.+\alpha_{\mathrm{nkc}-1}(l+1)\tau_{l,r}+\alpha_{\mathrm{nkc}-1}(-1)^{l}r\pi\tau_{l-1,r}\right\}\right]
\end{aligned}
$$

The horizontal velocity component u and the height function h as discussed in this appendix is used in case of an anti-symmetric flow about $x=0$ with only a no-slip condition at the bottom, see section 5.2. The program code is in file BvecXSin.cpp.

Appendix F

The treatment of the kinematic boundary condition for the case the velocities are expanded in Chebyshev polynomials

The horizontal velocity component u and the height h are expanded in Chebyshev polynomials $T_0, T_1, \cdots$ as follows:

$$u(x,y,t) = \sum_{i=0}^{\texttt{nic}-1} \sum_{k=0}^{\texttt{nkc}-1} d_{ik}(t) T_{2k+1}(x) \left(\frac{y}{h(x,t)} \right)^{i+1}$$

$$h(x,t) = \sum_{m=0}^{\texttt{Ncoef\_b}} b_m(t) T_{2m}(x)$$

The continuity equation and this expression for the velocity u yield the following expression for the vertical velocity component v:

$$\begin{aligned} v(x,y,t) = & \sum_{i=0}^{\texttt{nic}-1} \sum_{k=0}^{\texttt{nkc}-1} d_{ik}(t) T_{2k+1}(x) \frac{i+1}{i+2} \frac{\partial h}{\partial x} \left(\frac{y}{h(x,t)} \right)^{i+2} \\ & - \sum_{i=0}^{\texttt{nic}-1} \sum_{k=0}^{\texttt{nkc}-1} d_{ik}(t) T_{2k+1}(x) \frac{h(x,t)}{i+2} \left(\frac{y}{h(x,t)} \right)^{i+2} \end{aligned}$$

The next step is the substitution of the expansions in the kinematic boundary condition $\frac{\partial h}{\partial t} = v - u\frac{\partial h}{\partial x}$. Derivatives of Chebyshev polynomials appear in the resulting equation. These derivatives have to be expanded into Chebyshev polynomials, if possible. We therefore need the following lemma:

$$\frac{1}{n} \frac{\partial T_n(x)}{\partial x} = 2T_{n-1} + \frac{1}{n-2} \frac{\partial T_{n-2}(x)}{\partial x}$$

Proof:
Define $A = \arccos(x)$. Then $\sin^2 A + x^2 = 1$. Thus $\sin(A) = \sqrt{1-x^2}$. Note $T_n(x) = \cos(n\ A)$. Then

$$\frac{\partial T_n(x)}{\partial x} = \frac{n\ \sin(n\ A)}{\sqrt{1-x^2}} = \frac{n\ \sin(n\ A)}{\sin(A)}$$

And so

$$\begin{aligned}\frac{1}{n}\frac{\partial T_n(x)}{\partial x} - \frac{1}{n-2}\frac{\partial T_{n-2}(x)}{\partial x} &= \frac{\sin(n\ A) - \sin((n-2)\ A)}{\sin(A)} \\ &= \frac{2\cos((n-2)\ A)\ \sin(A)}{\sin(A)} = 2T_{n-1}\end{aligned}$$

■

This lemma shows

$$\frac{1}{n}\frac{\partial T_n(x)}{\partial x} = 2T_{n-1} + \frac{1}{n-2}\frac{\partial T_{n-2}(x)}{\partial x} = 2T_{n-1} + 2T_{n-3} + \frac{1}{n-4}\frac{\partial T_{n-4}(x)}{\partial x}$$

Induction for odd n yields

$$\frac{1}{n}\frac{\partial T_n(x)}{\partial x} = 2T_{n-1} + 2T_{n-3} + \cdots + 2T_2 + 1$$

Thus

$$\frac{\partial T_{2n+1}(x)}{\partial x} = (2n+1)(2T_{2n} + 2T_{2n-2} + \cdots + 2T_2 + 1)$$

We also need the following lemma

$$T_{m+n} + T_{m-n} = 2T_m T_n$$

This lemma immediately follows from the sum rule for cosines. ■The expansions are now substrituted in the kinematic boundary condition.

$$\begin{aligned} v - u\frac{\partial h}{\partial x} &= \sum_{i=0}^{\mathtt{nic-1}} \sum_{k=0}^{\mathtt{nkc-1}} \sum_{m=0}^{\mathtt{Ncoef\_b}} d_{ik} T_{2k+1} \frac{i+1}{i+2} b_m \frac{\partial T_{2m}}{\partial x} \\ &\quad - \sum_{i=0}^{\mathtt{nic-1}} \sum_{k=0}^{\mathtt{nkc-1}} \sum_{m=0}^{\mathtt{Ncoef\_b}} d_{ik} T_{2k+1} b_m \frac{T_{2m}}{i+2} \\ &\quad - \sum_{i=0}^{\mathtt{nic-1}} \sum_{k=0}^{\mathtt{nkc-1}} \sum_{m=0}^{\mathtt{Ncoef\_b}} d_{ik} T_{2k+1} b_m \frac{\partial T_{2m}}{\partial x} \\ &= -\sum_{i=0}^{\mathtt{nic-1}} \sum_{k=0}^{\mathtt{nkc-1}} \sum_{m=0}^{\mathtt{Ncoef\_b}} \frac{d_{ik}}{i+2} b_m \frac{\partial (T_{2k+1} T_{2m})}{\partial x} \\ &= -\frac{1}{2} \sum_{i=0}^{\mathtt{nic-1}} \sum_{k=0}^{\mathtt{nkc-1}} \sum_{m=0}^{\mathtt{Ncoef\_b}} \frac{d_{ik}}{i+2} b_m \left(\frac{\partial T_{2k+2m+1}}{\partial x} + \frac{\partial T_{|2k+1-2m|}}{\partial x} \right) \\ &= -\sum_{i=0}^{\mathtt{nic-1}} \sum_{k=0}^{\mathtt{nkc-1}} \sum_{m=0}^{\mathtt{Ncoef\_b}} \frac{d_{ik}}{i+2} b_m \left\{ (2k+2m+1) \sum_{l=0}^{k+m} T_{2l} \right. \end{aligned}$$

$$+|2k+1-2m| \sum_{l=0}^{\frac{|2k+1-2m|-1}{2}} T_{2l} + 2k + 2m + 1 + |2k+1-2m| \Bigg\}$$

Comparing coefficients of equal Chebyshev polynomials on both sides of this equation yields
for $r > 0$:

$$\frac{\partial b_r}{\partial t} =$$

$$-\sum_{k=0}^{\mathtt{nkc}-1} \left(\sum_{i=0}^{\mathtt{nic}-1} \frac{d_{ik}}{i+2} \right) \sum_{m=max(0,r-k)}^{\infty} (2k+2m+1) b_m$$

$$-\sum_{k=0}^{\mathtt{nkc}-1} \left(\sum_{i=0}^{\mathtt{nic}-1} \frac{d_{ik}}{i+2} \right) \sum_{m=0}^{k-r} (2k-2m+1) b_m$$

$$-\sum_{k=0}^{\mathtt{nkc}-1} \left(\sum_{i=0}^{\mathtt{nic}-1} \frac{d_{ik}}{i+2} \right) \sum_{m=k+r+1}^{\infty} (-2k+2m-1) b_m$$

and

$$\frac{\partial b_0}{\partial t} = -\sum_{k=0}^{\mathtt{nkc}-1} \left(\sum_{i=0}^{\mathtt{nic}-1} \frac{d_{ik}}{i+2} \right) \sum_{m=0}^{\infty} max(2k+1, 2m) b_m$$

The horizontal velocity component u and the height function h as discussed in this appendix is used in case of an anti-symmetric flow about $x = 0$ with only a no-slip condition at the bottom, see section 5.2. The program code is in file BvecCheb.cpp.

Appendix G

A model for the horizontal velocity component satifisfying the no-slip condition at three boundaries

Suppose a solid boundary occurs at $x = 1$ and the velocity component u is odd in x, i.e. $u(x, y, t) = -u(-x, y, t)$. The problem is to find a function f, see section 5.2 such, that the velocity component u satisfies the conditions. The conditions for the velocity component u are translated into conditions for the function f. The function f has to satisfy the following conditions for $l \geq 0$: 1. f is odd in x, i.e. $f(l, -x) = -f(l, x)$. 2. $f(l, 1) = 0$. 3. $\frac{\partial f}{\partial x}(l, 1) = 0$.

To prove condition 1, notice that $u(x, y, t) = -u(-x, y, t)$ for all possible choices of the coefficients d_{ik} in the expansion of u. Substitute $d_{0l} = 1$ and $d_{ik} = 0$ for $i \neq 0$ or $k \neq l$ into the equation $u(x, y, t) = -u(-x, y, t)$ and notice that $h(-x) = h(x)$.

Conditions 2 and 3 are consequences of the no-slip conditions $u(1, y, t) = 0$ and $v(1, y, t) = 0$. Substitution of $d_{0l} = 1$ and $d_{ik} = 0$ for $i \neq 0$ or $k \neq l$ into the equations $u(1, y, t) = 0$ and $v(1, y, t) = 0$ yields respectively $f(l, 1) = 0$ and $\frac{\partial f}{\partial x}(l, 1) = 0$.

The reverse is also true: if the function f satisfies the abovementioned three conditions, then u is odd in x and u satisfies the no-slip condition at $x = 1$. For the construction of a function f that satisfies the three boundary conditions, the solutions of the following characteristic value problem are used

$$\frac{d^4 y}{dx^4} = \alpha^4 y.$$

The boundary conditions are

$$y(-1) = y(1) = \frac{dy}{dx}(-1) = \frac{dy}{dx}(1) = 0$$

and y is odd. Let α_k denote a characteristic value and let y_k be the proper solution belonging to it. Then, multiplying the equation governing y_k by y_n (belonging to α_n) and integrating over the range of x, we obtain

$$\alpha_k^4 \int_{-1}^{1} y_k y_n dx = \int_{-1}^{1} y_n \frac{d^4 y_k}{dx^4} dx$$

Transforming the integral on the right-hand side by two successive integrations by parts, we are left with

$$\alpha_k^4 \int_{-1}^{1} \frac{d^2 y_n}{dx^2} \frac{d^2 y_k}{dx^2} dx,$$

the integrated parts vanishing (both times) on account of the boundary conditions on y_n. From the symmetry of the integral on the right-hand side of the last equation in n and k, it follows that

$$(\alpha_k^4 - \alpha_n^4) \int_{-1}^{1} y_k y_n dx = 0$$

Hence, for $k \neq n$

$$\int_{-1}^{1} y_k y_n dx = 0$$

The solutions y_k, therefore, form an orthogonal set. It can be shown that this set is complete, i.e. every continuous function in the interval $(-1, 1)$ which together with its first derivative vanishes at the endponts of the interval is a infinite linear combination of the elements of the set $\{y_n, n > 0\}$. The proper solutions of the fourth order differential equation and its boundary conditions are readily found.

$$y_k = f(x, k) = \frac{\sinh(\lambda_k x)}{\sinh(\lambda_k)} - \frac{\sin(\lambda_k x)}{\sin(\lambda_k)}$$

Defined in this manner, the function f clearly vanish at $x = 1$; that the derivative in x also vanish at this point requires that λ_k is the root of the characteristic equation

$$\coth(\lambda) + \cot(\lambda) = 0.$$

It can be readily verified that function $f(x, k)$ satisfies

$$\int_{-1}^{1} f(x, k) f(x, n) dx = \delta_{kn}.$$

For $k > 4$ the asymptotic formula

$$\mu_k \sim (2k + \frac{1}{2}) \frac{\pi}{2}$$

gives the roots correct to ten decimals. Further, $\mu_1 = 3.92660231, \mu_2 = 7.06858274, \mu_3 = 10.21017612, \mu_4 = 13.35176878$.

Now suppose that a solid boundary occurs at $x = 0$ and $x = 1$. The problem is to find a function f such, that the velocity component u satisfies the no-slip conditions. The conditions for the function f now read for $l \geq 0$: 2. $f(l, 0) = f(l, 1) = 0$. 3. $\frac{\partial f}{\partial x}(l, 0) = 0$, $frac\partial f \partial x(l, 1) = 0$.

Consider the characteristic value problem defined by the equation

$$\frac{d^4y}{dx^4} = \alpha^4 y.$$

and the boundary conditions

$$y(0) = y(1) = \frac{dy}{dx}(0) = \frac{dy}{dx}(1) = 0$$

The proper solutions of these equations fall into two non-combining groups

$$y_k = f_1(x,k) = \frac{\cosh(\lambda_k(2x-1))}{\cosh(\lambda_k)} - \frac{\cos(\lambda_k(2x-1))}{\cos(\lambda_k)}$$

and

$$y_k = f_2(x,k) = \frac{\sinh(\mu_k(2x-1))}{\sinh(\mu_k)} - \frac{\sin(\mu_k(2x-1))}{\sin(\mu_k)}$$

These functions clearly vanish at $x = 0$ and $x = 1$. Their derivatives also vanish at these points if

$$\tanh(\lambda) + \tan(\lambda) = 0$$

and

$$\coth(\mu) - \cot(\mu) = 0$$

The functions f_1 and f_2 have the property

$$\int_0^1 f_i(x,k) f_j(x,n) dx = \delta_{ij}\delta_{k,n}$$

For $k > 4$ the asymptotic formulae

$$\lambda_k \sim (2k - \frac{1}{2})\frac{\pi}{2}$$

and

$$\mu_k \sim (2k + \frac{1}{2})\frac{\pi}{2}$$

give the roots correct to ten decimals. Further, $\lambda_1 = 2.36502037$, $\lambda_2 = 5.49780392$, $\lambda_3 = 8.63937983$, $\lambda_4 = 11.78097245$ and $\mu_1 = 3.92660231$, $\mu_2 = 7.06858274$, $\mu_3 = 10.21017612$, $\mu_4 = 13.35176878$.

Appendix H

Code of programme

The code of the Marangoni programme of version 3 (3λ's) are given below for the four different expansion: normal sin (bndr1.cpp), chakandrasekar (bound.cpp), Chebyshev (bvec-cheb.cpp), and sin with extra linear term (xsin.cpp). Marangoni.cpp, Timestep.cpp, Newuhdb.cpp, Gvector.cpp, Ftr.cpp and Gee3.cpp are general to use.

```
/*

Program Marangoni_version_12;



  This program has been developed to compute the motion of a liquid layer
  caused by the Marangoni effect. This means that there arise velocities
  in the considered liquid layer by variations in the surface stress. This
  causes a transformation of the liquid surface. When this effect arises
  at a heat exchanger, the heat transfer will be increased.

 * Index #         : 3
 * Title           : Marangoni.cpp
 * Last Edit       : August 3 1993
 * Copyright       : CMHT
 * System          : C++
 * Description     : Computation of the velocities in a thin liquid layer
                     caused by the Marangoni effect.
 * Date 1st issue  : October 19 1992
 * History         : Originally written in Turbo Pascal
 * Status          : -
_________________________________________________________________________*/

#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <math.h>
#include <fstream.h>
#include "Definitions.hpp"
#include "MaraVar.hpp"
#include "Basics.hpp"
#include "RunScr.hpp"
#include "c_Computation.hpp"
#include "Init_New_Run.hpp"
#include "EnterData.hpp"
#include "Data.hpp"
#include "NewNkNi.hpp"
#include "TimeStep.hpp"
#include "gvector.hpp"

//**************class Basics****************

double Sqr(double r)
{
 return r*r;
}

ostream& cleol(ostream& os)
{
      clreol();
      return os;
}

ostream& xy(ostream& os, Point p)
{
      gotoxy(p.x, p.y);
      return os;
}

OMANIP(Point) xy(int x, int y)
{
      Point p;
      p.x = x; p.y = y;
      return OMANIP(Point)(xy, p);
}

//**************class MaraVar***************

int MaraVar::ni=3;                        int MaraVar::nkc=3;
int MaraVar::nk=3;                        int MaraVar::nic=3;
int MaraVar::ntimesteps=1;                int MaraVar::par=0;
int MaraVar::Ncoef_b=3;                   int MaraVar::step;
double MaraVar::surft1=72.3e-3;           double MaraVar::delsig=49.5e-3;
double MaraVar::surft2=22.8e-3;           double MaraVar::timestep=0.01;
double MaraVar::sx[nkmax+1];              double MaraVar::sgr_xis1;
double MaraVar::b[10*ncoefmax];           double MaraVar::time=0;
double MaraVar::c[nimax];                 double MaraVar::x[nkmax];
double MaraVar::sgr[nkmax+1];             double MaraVar::dsxdt[nkmax];
double MaraVar::re;

//*************class Data : public virtual MaraVar *******************

double Data::d3hdtdx2[nkmax];          double Data::d2hdx2[nkmax];
double Data::dhdt[nkmax];              double Data::d2hdxdt[nkmax];
double Data::hx[nkmax];                double Data::d3hdx3[nkmax];
double Data::uc[nkmax];                double Data::dhdx[nkmax];

int Data::OddNeg(int p)
{
 int q;
 q=2*(p/2);
 return p==q?1:-1;
}

//**************class OutFile : public virtual MaraVar************

void InOutFile::OutStream()
{
 int i;

 ofstream output("MaranOut");
 output << nk << " " << ni << " " << ntimesteps << " " << nkc
      << " " << nic << " " << Ncoef_b << " " << surft1 << " "
      << surft2 << " " << time << endl;
 for(i=0;i<nkc*nic;i++) output << d[i] << " ";
 output << endl;
 for(i=0;i<=Ncoef_b;i++) output << b[i] << " ";
 output << endl;
 for(i=0;i<nkc*nic;i++) output << d_old[i] << " ";
 output << endl;
 for(i=0;i<nkc*nic;i++) output << e_old[i] << " ";
}

void InOutFile::InStream()
{
 int i;

 ifstream input(*filename);
 if(!input){
   cout << "\a";
   c123=0;
   *filename = "h";}
 else{
     input >> nk >> ni >> ntimesteps >> nkc >> nic >> Ncoef_b >> surft1
           >> surft2 >> time;
     for(i=0;i<nkc*nic;i++) input >> d[i];
     for(i=0;i<=Ncoef_b;i++) input >> b[i];
     for(i=0;i<nkc*nic;i++) input >> d_old[i];
     for(i=0;i<nkc*nic;i++) input >> e_old[i];
     par = 2;
 }
 delsig = surft1 - surft2;
}

//**************class RunScr : public virtual MaraVar************

void RunScr::RunScreen(int j)
{
 if(j==1){
   cout << xy(48,2) << "     " << xy(48,2) << step;
   }

   else{
       cout << xy(6,1)  << "ni " << ni << xy(6,2) << "nk " << nk
      << xy(15,1) << "nic " << nic << xy(15,2) << "nkc "
      << nkc << xy(24,1) << "Ncoef_b " << Ncoef_b << xy(38,1)
      << "timesteps " << ntimesteps << xy(38,2) << "step      " << step
      << setiosflags(ios::showpoint | ios::fixed)
      << xy(53,1) << "surft1  " << setprecision(4)
      << setw(6) << surft1 << xy(53,2)
      << "surft2  " << setw(6) << surft2;
   };
}

//***********************************************************************
```

```
//*********************************************************************

/**************************************************************************
*
 *
*
 *                              SxSgrComputation
*
 *
*
 *************************************************************************
*/

void TimeStep::SxSgrComputation()
{
 int f;
 double epower, betha, help, ndef, gamma;

 sgr_xis1 = surft1/delsig;
 if(delsig!=0){
   betha  = sqrt(2.0)/2;
   if(time<= betha && time!= 0){
     gamma = 4.0 / Sqr(betha * betha);
     help  = Sqr(time*(time- betha));
     for(f=0;f<nk;f++){
          ndef      = sqrt(1 + Sqr(aver * dhdx[f]));
          epower    = pow(1.0-x[f],betha/time);
          sx[f]     = sin(pi*(1.0-epower))*pi/time*epower/(1.0-x[f])*gamma
                      *betha;
          dsxdt[f] = sx[f] * gamma*time*(time-betha)*(2.0*time-betha)+
                      (-pi*pow(1.0-x[f],-(-betha+1.0*time)/time)*(-2.0*
                      cos(pi-pi*pow(1.0-x[f],betha/time))*pi*
                      pow(1.0-x[f],betha/time)*log(1.0-x[f])+2.0*
                      sin(pi-pi*pow(1.0-x[f],betha/time))*log(1.0-x[f])
                      +sin(pi-pi*pow(1.0-x[f],betha/time))*sqrt(2.0)*time)
                      /(time*time*time)/2)*2.0*gamma*help;
          sx[f]    *= lengt * help;
          sgr[f]    = sgr_xis1 - gamma * (1.0 + cos(pi*(1.0-epower))) * hel
p;
          dsxdt[f]  = lengt * dsxdt[f] / ndef - sx[f] * aver * aver * dhdx[
f] *
                       d2hdxdt[f] / (ndef * ndef * ndef);
          sx[f]    /= ndef;
     }
     sx[nk]    = 0.0;
     sgr[nk]   = sgr_xis1;
/*     sx[nk]    = sx[nk-1];
     sgr[nk]   = sgr[nk-1];*/

   }
   if(time> betha)
     for(f=0;f<nk;sgr[f]=sgr_xis1,sx[f]=dsxdt[f]=0,f++);
 }
 if(delsig==0){  // Tangential stress condition at the interface is zero.
   for(f=0,sgr_xis1=surft1;f<nk;sgr[f]=surft1,f++);
 }
}
//-------------------------- End of SxSgrComputation---------------------
-

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

/***********************************************************************
*
 *
*
 *                              INIT_NEW_RUN
*
 *
*
  **********************************************************************
**/

// Initialisation of the protected members of class MaraVar

//***********class Init_New_Run : public c_Computation ******

void Init_New_Run::IncorrIndex()
{
 char cr;
 cout << xy(5,20) << "\aIncorrect array index."
      << xy(5,22) << "Ready (y,n)";
      cr=getch();
      if(cr=='0') tick=0;
      if(cr=='1') tick=1;
      if(cr=='a') tick=2;
      if(cr=='y') ready=1;
 cout << xy(5,20) << cleol << xy(5,22) << cleol;
}

void Init_New_Run::EnterIndex()
{
 int n2;
 char cr;

 k=0;
 for(;;){
    cr=getche();
    n2=cr-'0';
    if(n2==-16){    // n2==-16 is <Spacebar>
      tick=1-tick;
      if(tick==1) cout << xy(5,12) << cleol;
    }
    if(cr=='a'){cout << " \b"; tick=2;};
    while(n2==-40){ // n2==-40 is <Backspace>
         cout << " \b";
         k=k/10;
         cr=getche();
         n2=cr-'0';
    }
    if(n2<0 || n2>9) break;
    k=10*k+n2;
 }
}

void Init_New_Run::ScrnQuest(char a)
{
 int n;

 tick=1;
 n3=5;
 clrscr();
 cout << xy(5,10) << "Do you want to change the " << a <<
        " coefficients (y/N/a) ?";
 ch = getch();
 n=ch-'0';
 while(ch!='y' && ch!='n' && ch!='a' && n!=-35){    // n==-35 is <Enter>
      cout << xy(5,10) << cleol <<
              "\aEnter 'y', 'n' or 'a' or press <Enter> (='n')";
      ch = getch();
      n=ch-'0';
 }
 if(ch=='a'){tick=2;}
 else{
     cout << xy(2,5) << "Changed array-indices: ";
     n1 = wherex();
     n1++;
 }
}

/***********************************************************************
*
 *
*
 *                              BandDadjustment
*
 *
*
 ***********************************************************************
*

 With this procedure the initial values of the b and d coefficients can be
 adjusted. By this we can study the effects of changing one or some of the
 coefficients b and d on the calculations and the plots of the marangoni
 flows.
*/

void Init_New_Run::BandDAdjustment()
{
 char yn;
 int ik,i,n;
```

```
  clrscr();
  cout << xy(5,10) << "Will there be a driving force? (Y/n)";
  yn = getch();
  if(yn == 'n') time = 100;
  cout << xy(5,10) << cleol;

  ScrnQuest('b');
  if(ch=='y' || ch=='a'){
    par=1;
    ready=0;
    while(!ready){
         if(tick!=2){
           cout << xy(5,10) << cleol << "Enter the array-index (at most "
                << Ncoef_b <<")  ";
           EnterIndex();
         }
     if(k >= 0 && k <= Ncoef_b){
       if(ch=='y'){
         cout << xy(n1,n3) << k;
         n1=wherex();
         n1++;
         if(n1>=70){n1=26;++n3;};
       }
       cout << xy(5,10) << cleol << "Enter the value of b[" << k << "]  ";
       if(tick==1 || tick==2){
         cin >> b[k];
         if(tick==2){
               if(k==Ncoef_b) ready=1;
               k++;
         }
       }
       else{
             cin >> b[k];
             cout << xy(5,12) << "Ready (y/n) ";
             ch=getch();
             if(ch == 'y') ready=1;
       }
     }
     else IncorrIndex();
    }
  }
  ScrnQuest('d');
  i=0;
  if(ch == 'y' || ch == 'a'){
    par=1;
    ready=0;
    while(!ready){
         if(tick!=2){
           cout << xy(5,10) << cleol << "Enter the array-indices i and k"
                << xy(5,12) << "i ( between 0 and " << nic-1 << " ): " << cl
eol;
           EnterIndex();
           i=k;
```

```
          cout << xy(5,13) << "k ( between 0 and " << nkc-1 << " ): " << cl
eol;
          EnterIndex();
        }
        if(k<nkc && i<nic && k>=0 && i>=0){
          if(ch == 'y'){
            cout << xy(n1,n3) << i << "," << k;
            n1=wherex();
            n1++;
            if(n1>=70){n1=26;++n3;};
          }
          cout << xy(5,14) << "Enter the value of d[" << i << "," << k << "
] "
               << cleol;
          ik = i * nkc + k;
          if(tick==1 || tick==2){
            cin >> d[ik];
            cout << xy(5,16) << cleol;
            if(tick==2){
              ++k;
              if(k==nkc){
            k=0;i++;
            if(i==nic) ready=1;
          }
        }
      }
          else{
          cin >> d[ik];
          cout << xy(5,16) << "Ready (y/n)";
          ch=getch();
          if(ch=='y') ready=1;
      }
    }
    else IncorrIndex();
   }
   for(i=0;i<nic*nkc;i++) d_old[i] = d[i];
 }

 clrscr();
 cout << "\nB values:\n";
 for(k=0;k<=Ncoef_b;k++){
    cout << b[k] << "  ";
    i=10*((k+1)/10);
    if (k==i && k!=0) cout << "\n";
 }
 cout <<  "\n\n\nD values:\n";
 for(i=0;i<nic;i++){
    for(k=0;k<nkc;k++){
       ik = i * nkc + k;
       cout << setprecision(2) << setiosflags(ios::right) << setfill(' ')
            << setw(4) << d[ik] << "  ";
    }
    cout << "\n";
    cout << resetiosflags(ios::right);
 }
 cout << "\Enter 'r' to continue entering data";
 cout << "\nPress enter to continue...";
 ch=getch();
 if(ch=='r') BandDAdjustment();
}

//*************class c_Computation : public virtual MaraVar**********

void c_Computation::Zero_Calc(double* y,int l)
{
 int i,j;
 double z1,z,pp,p3,p2,p1;

 for(i=1;i<=l;i++){
    z=cos(2*(pi*i)/(l+0.5));
    do{
      for(p1=1.0,p2=0.0,j=1;j<=l;j++){
         p3=p2;
         p2=p1;
         p1=((2.0*j-1.0)*z*p2-(j-1.0)*p3)/j;
      };
      pp=l*(z*p1-p2)/(z*z-1.0);
      z1=z;
      z=z1-p1/pp;
    }
    while(fabs(z-z1)>1.0e-10);
    y[i]=0.5*(1-z);
 };
}

//---------------------End of Zero_Calc-------------------------------------

void c_Computation::c_Comp(double* c,int jx)
{
 int i,j;
 double s;
 Zero_Calc(c,jx);
 for(j=1;j<=jx;j++)
    for(i=0;i<=jx-j;i++)
       if(c[i+1]>c[i]){
         s=c[i];
         c[i]=c[i+1];
         c[i+1]=s;
       }
}

//--------------------------End of c_Comp-----------------------------------
-

void Init_New_Run::StartNew()
{
// c_Computation::c_Comp(x,nk);
// c_Computation::c_Comp(c,ni);
```

```
 for(int j=0;j<nk;j++) x[j] = 1.0*(nk-j)/(nk+1);
 for(int l=0;l<ni;l++) c[l] = 1.0*(ni-l)/(ni+1);

// Initialisation of the first b value =>
 if(par==0) b_init();

 BandDAdjustment();    // In order to provide new starting
                       // values for the b and d coefficients

 if(delsig != 0){
    // driving force for the tang.stress cond. and so for marangoni flows
    re      = delsig * aver * thickn * rho / (mic * mic);
 }
 else{
     // Make believe that delsig = 1 because of dimensionlessness
     re      = aver * thickn * rho / (mic * mic);
 }
}

//***************class EnterData : public InOutFile***********

void EnterData::StartScreen()
{
 clrscr();
 highvideo();
 cout << xy(15,7) << "INPUT STARTING DATA";
 lowvideo();

 if (fn==1){
    cout << xy(15,22) << "For entering data by hand  type h ";
    cout << xy(15,23) << "For the default input file type x ";
    cout << xy(15,24) << "For another input file     type <filename>";
 }

 cout << xy(15,10)
      << "1) Number of x collocation points, nk : " << nk << xy(15,11)
      << "2) Number of y collocation points, ni : " << ni << xy(15,12)
      << "3) Number of time steps               : " << ntimesteps << xy(15,
13)
      << "4) Number of columns of the matrix d  : " << nkc << xy(15,14)
      << "5) Number of rows of the matrix d     : " << nic << xy(15,15)
      << "6) Number of height coefficients b    : " << Ncoef_b << xy(15,16)
;
 if (sw==1){ cout << "7) Surface stress liquid " << st+1
      << "                : "; if(st==0){cout << surft1;} else cout << surft2;}
 else        cout << "7) Surface stresses "; cout << xy(15,17)
      << "8) Enter data or name of datafile     : " << *filename << xy(15,1
8)
      << "9) Ready" << xy(15,20)
      << "Select an item: ";

}
```

```
/***************************************************************************
*
 *
*
 *                              ReadData
*
 *
*
 ***************************************************************************
*

  Reads the starting data from the screen. You may enter nk, ni, nkc and ni
 c,
  and the number of time steps as well as the surface stresses s1 and s2.
*/

void EnterData::ReadData()
{
 int n;
 char ch,u;

 ofstream output("outdata");
 while(!ready){
      StartScreen();
      ch = getche();
      n = ch-'0';
      if (n==-35) n=10;
      if (n==7 && sw==0){ sw=1; StartScreen(); };
      if (n==8){ fn=1; StartScreen(); fn=0; };
      while (c123==1 && n>3 && n<8){ cout << "\a\a"; StartScreen();
            ch = getche(); n = ch-'0'; if (n==-35) n=0; };
      cout << xy(55,9+n) << cleol;
      switch (n){
        case 1 :  cin >> nk ; if(nk>nkmax){cout << "\a"; nk = nkmax;}; brea
k;
        case 2 :  cin >> ni ; if(ni>nimax){cout << "\a"; ni = nimax;}; brea
k;
        case 3 :  cin >> ntimesteps ; break;
        case 4 :  cin >> nkc ; if(nkc>nkcmax){cout << "\a"; nkc = nkcmax;};
 break;
        case 5 :  cin >> nic ; if(nic>nicmax){cout << "\a"; nic = nicmax;};
 break;
        case 6 :  cin >> Ncoef_b ; break;
        case 7 :  cin >> sf ; if(st==0){ surft1 = sf ; delsig = sf - surft2
;}
                          else { surft2 = sf ; delsig = surft1 - sf;};
                          st = 1 - st; break;
        case 8 :  cin >> *filename; u = *filename[0];
                  if(u!='h'){
                    c123=1; if(u=='x') *filename = "MaranOut";
                    InStream();
                  }
                  else c123=0; break;
        case 9 :
```

```
        case 10:  ready = 1 ; output << "nk=" << nk << " ni=" << ni
                  << " ntimesteps=" << ntimesteps << " nkc=" << nkc
                  << " nic=" << nic << " Ncoef_b=" << Ncoef_b << " surft1="
                  << surft1 << " surft2=" << surft2 << endl; break;
      }
 }
}

//***class NewNkNi : public c_Computation, public hGradient***

void NewNkNi::NewNkNi_uc_h()
{
 int f,i,k,ik;
 double h2;

 hGradientCalculation();

 for(f=0;f<nk;f++){
    uc[f] = 0.0;
    for(k=0;k<nkc;k++){
       h2 = fc0(k,x[f]);
       for(i=0;i<nic;i++){
          ik = i * nkc + k;
          uc[f] += d[ik] * h2;
       }
    }
 }
}

//---------------------- End of NewNkNi_uc_h ------------------------------


 TimeStep tmestep;
 NewNkNi new_nk_ni_uc_h;
 Data data;
 EnterData screen;
 Init_New_Run initnewrun;
 MaraVar mv;
 RunScr run;
 InOutFile out;

void main()
{
 int beginstep;
 int help;

 help=0;
 screen.ReadData();
 initnewrun.StartNew();
 clrscr();

 highvideo();
 cout << xy(1,12) << cleol << xy(32,12) << "contour integration";
 cout << xy(36,14) << "t=" << xy(43,14) << " y=";
 lowvideo();

 beginstep = mv.stp();
 while(mv.stp() < mv.ntmstps()+beginstep){
      mv.plusstep();
      run.RunScreen(help);
      help=1;
      new_nk_ni_uc_h.NewNkNi_uc_h();
      if(beginstep==mv.stp()-1) tmestep.SxSgrComputation();
      tmestep.ATimeStep();
 }
 out.OutStream();
 clrscr();
 cout << " Index " << 3 << "  Name: Marangoni.cpp " << " version 12" << end
l;
}
```

```
#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <math.h>
#include <fstream.h>
#include "Definitions.hpp"
#include "Basics.hpp"
#include "Data.hpp"
#include "Contour_Int_New.hpp"
#include "New_DB.hpp"
#include "TimeStep.hpp"
#include "SolveWTT_Cond.hpp"
#include "gvector.hpp"
#include "matrix.hpp"
#include "vector.hpp"

//*********class Functions : public virtual Data*********

/***************************************************************************
  ** Function dvdy                                                       **
  *************************************************************************
*/

//Here the analytical expression for dv/dy is calculated

double Functions::funct_dvdy(double x,double c,
double hx,double dhdx)
{
 int i,k;
 register double help,h0,h1,h4;

 h4=dhdx/hx;
 for(help=0.0,k=0;k<nkc;k++){
    h0 = fc0(k,x) * h4;
    h1 = -fc1(k,x);
    for(i=0;i<nic;i++) help += d[i*nkc+k] * (h1*gc0(i,c) + h0*c*gc1(i,c));
 }
 return help;
}

//---------------------- End of funct_dvdy -------------------------------
-

/**************************************************************************
  ** Function v                                                          **
  *************************************************************************
/
// Here the velocity v is computed.

double Functions::funct_v(double x,double c,double hx,double dhdx)
{
 int i,k;
 register double help,h0,h1;
 for(help=0.0,k=0;k<nkc;k++){
    h0 = fc0(k,x) * dhdx;
    h1 = -hx*fc1(k,x) - h0;
    for(i=0;i<nic;i++) help += d[i*nkc+k] * (h0*c*gc0(i,c) + h1*gg(i,c));
 }
 return help;
}

//------------------------- End of funct_v ------------------------------
-

/*********************************************************************
***
 ***  Procedure ContourInt computes the integrals and the matrix elements
***
 ***
***
 ***
***
 ***          (P1-P2) + (P2-P3) + (P3-P4) + (P4-P0) + (P0-P1) = 0
***
 ********************************************************************
***

Before solving the time evolution, the Navier-Stokes equations have to be
solved. Consequently a great number of integrals have to be solved.
In fact, for every collocation point there has to be integrated along a
contour from the collocation point to the wall, along a line of constant
c, from there to the surface of the liquid and along a vertical line back t
o
the collocation point.
*/

//**class TimeStep : public Contour_Int_New, public New_DB, public NewNkNi*
*

void TimeStep::ContourInt(matrix & ah)
{
 /*
   Here the matrix ah and the vector bvec are composed for the computation
   of the time derivates of the d values, namely the e values, for the
   collocation points situated between 0<x<1 and 0<y<1.
  */

 int ff,q,qc,g,i,j,k,ik,c2,z2;

 double sqraver,Iy_dvdt_xis1,Iy_udvdx_xis1,Iy_d2vdx2_xis1,h_xis1,dhdx_xis1,
        d2hdx2_xis1,d3hdx3_xis1,diffc,h1,delc,cv,x1,cvc,alfap_1,alfap_2,
        ssum1,ssum2,ssum3,ssum4,fg,fxg,fxxg,fgc,fgg,fxgg,fxxgg,fgg1,p0_p1,
        fxgg1,fxgc,fgcc,fxxgg1,fxxxgg,hp0,hp1,hp2,hp3,hp4,hp5,P_dvdx,p1_p2,
        P_dhdt,P_d2hdxdt,P_v,Iy_d2v_dy2,coprex,Iy_vdvdy,p34,Ix_vdudy,p2_p3,
        Ix_dhdx_vdvdy,Ix_dhdx_udvdx,Ix_dhdx_d2vdx2,Ix_d2u_dx2,Ix_d2u_dy2,
        Ix_dhdx_d2vdy2,Ix_dudt,Ix_dhdx_dvdt,diff,xi,hh,dd,d2,d3,alfap_4,
        Ix_ududx,P_dudx,p3_p4,p4_p0,p4_p1;

 register double rd,norm,nor;

 vector Iy_udvdx(nk),Iy_dvdt(nk),Iy_d2vdx2(nk),i11(nic),i12(nic),sums1(nic)
,
        sums2(nic),sums3(nic),sums4(nic),sum1(nkc),sum2(nkc),sum3(nkc),
        sum4(nkc),sum5(nkc),kk1(nkc),kk2(nkc),kk3(nkc);

 matrix ii1(nk,nic),ii2(nk,nic),alfap_3(nic,nkc);

 cout << xy(10,9) << cleol;
 cout << xy(39,14) << "   " << xy(39,14) << ntimesteps-step+1;

 sqraver = aver * aver;

 Iy_dvdt_xis1 = Iy_udvdx_xis1 = Iy_d2vdx2_xis1 = 0;

 h_xis1 = funct_h(1.0);
 dhdx_xis1 = funct_dhdx(1.0);
 d2hdx2_xis1 = funct_d2hdx2(1.0);
 d3hdx3_xis1 = funct_d3hdx3(1.0);

 for(i=0;i<nk;i++) Iy_udvdx[i]  = Iy_dvdt[i] = Iy_d2vdx2[i] = 0.0;
 for(ff=0;ff<nk;ff++) for(i=0;i<nic;i++) ii1(ff,i) = ii2(ff,i) = 0;
 for(i=0;i<nic;i++) i11[i] = i12[i] = 0;

 for(k=0;k<=nkc*nic;k++)
    for(i=k;i<=nic*nkc;i++)
       ah(k,i) = 0.0;

 for(g=0;g<ni;g++){

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

   cout << xy(47,14) << "  " << xy(47,14) << ni-g;

 // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

   if(g==0)
     diffc = 1-c[g];
   else
     diffc = c[g-1] - c[g];            // distance between succeeding c-val
ues.
   h1 = part * diffc / (1 - c[ni-1]); // h1 is a help variable; the interv
al
   c2 = int(h1); c2 += (1 - OddNeg(c2)) / 2; // is devided into c2 parts
   if(c2<4) c2 = 4;
   delc = diffc / c2;                  // the step size
   cv = c[g] + diffc;                  // the actual c-value

 //-- calculation of p4-p3 ---

   for(i=0;i<nic;i++){
```

2

```
      ik = i * nkc - 1;
      for(ssum1=ssum2=ssum3=ssum4=0,k=0;k<nkc;k++){
         ik++;
         ssum1 += d[ik] * fc0(k,1);
         ssum2 += d[ik] * fc1(k,1);
         ssum3 += d[ik] * fc2(k,1);
         ssum4 += d[ik] * fc3(k,1);
      }
      sums1[i] = ssum1;
      sums2[i] = ssum2;
      sums3[i] = ssum3;
      sums4[i] = ssum4;
   }
   x1 = dhdx_xis1 / h_xis1;
   cvc = cv;
   qc=5;
   for(j=0;j<=c2;j++){
      qc=6-qc;
      if(j==c2) qc=1;
      nor=qc*h_xis1*delc/3;
      fg = fxg = fxxg = fgc = fgg = fxgg = fxxgg = fgg1 = fxgg1 = fxgc =
      fgcc = fxxgg1 = fxxxgg = 0;
      for(i=0;i<nic;i++){
         hp0     = gc0(i,cvc);
         hp1     = gc1(i,cvc);
         hp2     = gc2(i,cvc);
         hp3     = gc3(i,cvc);
         hp4     = gg(i,cvc);
         hp5     = gg(i,1);
         fg     += sums1[i] * hp0;
         fxg    += sums2[i] * hp0;
         fxgc   += sums2[i] * hp1;
         fxxg   += sums3[i] * hp0;
         fgc    += sums1[i] * hp1;
         fgcc   += sums1[i] * hp2;
         fgg    += sums1[i] * hp4;
         fxgg   += sums2[i] * hp4;
         fxxgg  += sums3[i] * hp4;
         fxxxgg += sums4[i] * hp4;
         fgg1   += sums1[i] * hp5;
         fxgg1  += sums2[i] * hp5;
         fxxgg1 += sums3[i] * hp5;
         i11[i] += nor * cvc * hp0;
         i12[i] += nor * hp4;
      }
      P_dvdx          = cvc * (2 * dhdx_xis1 * fxg + d2hdx2_xis1 * fg - cvc
 * fgc *
                        dhdx_xis1 * x1) -(h_xis1 * fxxgg + 2 * dhdx_xis1 *
fxgg + d2hdx2_xis1 * fgg);
      Iy_udvdx_xis1 += nor * fg * P_dvdx;
      P_dhdt          = -h_xis1 * fxgg1 - dhdx_xis1 * fgg1;
      P_d2hdxdt       = -2 * dhdx_xis1 * fxgg1 - h_xis1 * fxxgg1 - d2hdx2_x
is1 * fgg1;
      P_v             = cvc * fg * dhdx_xis1 - h_xis1 * fxgg - dhdx_xis1 *
```

```
fgg;
       Iy_dvdt_xis1  += nor * ((cvc * (-cvc * x1 * fgc + fxg)
                        - fxgg) * P_dhdt + (-fgg + cvc * fg) *
                        P_d2hdxdt);
       Iy_d2vdx2_xis1+= nor * (cvc * (3 * dhdx_xis1 * fxxg - 3 * cvc
                        * dhdx_xis1 * x1 * fxgc + 3 * d2hdx2_xis1 * fxg - 3
 * cvc * d2hdx2_xis1
                        * x1 * fgc + d3hdx3_xis1 * fg + dhdx_xis1 * x1 * x1
 * cvc * (3
                        * fgc + cvc * fgcc)) - (h_xis1 * fxxxgg + 3 * dhdx_
xis1
                        * fxxgg + 3 * d2hdx2_xis1 * fxgg + d3hdx3_xis1 * fg
g));
       cvc -= delc;
       if(j==0) qc=2;
    }

    //--- calculation of p4-p3 ---
    Iy_d2v_dy2 = funct_dvdy(1,1,h_xis1,dhdx_xis1)
                 - funct_dvdy(1,c[g],h_xis1,dhdx_xis1);
    coprex     = Sqr(funct_v(1,1,h_xis1,dhdx_xis1));
    Iy_vdvdy   = funct_v(1,c[g],h_xis1,dhdx_xis1);
    Iy_vdvdy   = 0.5*(coprex-Sqr(Iy_vdvdy));

    //*** (p4-p3)***

    coprex = sqraver * Iy_d2v_dy2 + Sqr(sqraver) * Iy_d2vdx2_xis1;
    p34    = -coprex + re*sqraver*aver * (Iy_dvdt_xis1 +
             Iy_udvdx_xis1 + Iy_vdvdy);

//---------------------------------------------------------------------------
    for(k=0;k<nkc;k++){
       sum1[k] = sum2[k] = sum3[k] = sum4[k] = sum5[k] = 0;
       for(i=0;i<nic;i++){
          ik = i * nkc + k;
          sum1[k] += d[ik] * gc0(i,c[g]);
          sum2[k] += d[ik] * gc1(i,c[g]);
          sum3[k] += d[ik] * gg(i,1);
          sum4[k] += d[ik] * gg(i,c[g]);
          sum5[k] += d[ik] * gc2(i,c[g]);
       }
    }

    for(ff=0;ff<nk;ff++){
       for(k=0;k<nkc;k++) kk1[k]=kk2[k]=kk3[k]=0;
       Ix_ududx        = 0;
       Ix_vdudy        = 0;
       Ix_dhdx_vdvdy   = 0;
       Ix_dhdx_udvdx   = 0;
       Ix_dhdx_d2vdx2  = 0;
       Ix_d2u_dx2      = 0;
       Ix_d2u_dy2      = 0;
       Ix_dhdx_d2vdy2  = 0;
       Ix_dudt         = 0;
```

```
        Ix_dhdx_dvdt    = 0;
        q=5;
        if(ff==0)
          diff = 1-x[ff];
        else
          diff = x[ff-1] - x[ff];   // distance between succeeding x-values.
        h1 = part * diff / (1 - x[nk-1]); // h1 is a help variable; the inte
rval
        z2 = int(h1);                  // is devided into z2 parts
        z2 += (1 - OddNeg(z2)) / 2;
        if(z2<4) z2 = 4;
        for(j=0;j<=z2;j++){
           xi = x[ff] + diff - j*diff/z2;      // the actual x-value
           q=6-q;
           hh = funct_h(xi);
           dd = funct_dhdx(xi);
           d2 = funct_d2hdx2(xi);
           d3 = funct_d3hdx3(xi);
           if(j==z2) q=1;
           norm = q * diff / z2 / 3;
           x1=dd/hh;

           fg = fxg = fgc = fgg = fxgg = fxxgg = fxxg = fxgc = fxxxgg =
           fgg1 = fxgg1 = fxxgg1 = fgcc = 0;
           for(k=0;k<nkc;k++){
              hp0      = fc0(k,xi);
              hp1      = fc1(k,xi);
              hp2      = fc2(k,xi);
              hp3      = fc3(k,xi);
              fg      += sum1[k] * hp0;
              fxg     += sum1[k] * hp1;
              fgc     += sum2[k] * hp0;
              fgcc    += sum5[k] * hp0;
              fxxxgg  += sum4[k] * hp3;
              fxgc    += sum2[k] * hp1;
              fgg     += sum4[k] * hp0;
              fxgg    += sum4[k] * hp1;
              fxxgg   += sum4[k] * hp2;
              fxxg    += sum1[k] * hp2;
              fgg1    += sum3[k] * hp0;
              fxgg1   += sum3[k] * hp1;
              fxxgg1  += sum3[k] * hp2;
              kk1[k]  += norm * hp0;
              kk2[k]  += norm * dd * dd * hp0;
              kk3[k]  += norm * hp1 * hh * dd;
           }
           P_dvdx          = c[g] * (2 * dd * fxg + d2 * fg - c[g] * fgc *
                             dd * x1) -(hh * fxxgg + 2 * dd * fxgg + d2 * fg
g);
           P_dudx          = fxg - fgc * c[g] * x1;
           P_v             = c[g] * fg * dd - (hh * fxgg + dd * fgg);
           P_dhdt          = -hh * fxgg1 - dd * fgg1;
           P_d2hdxdt       = -2 * dd * fxgg1 - hh * fxxgg1 - d2 * fgg1;
           Ix_ududx       += norm * fg * P_dudx;
```

```
            Ix_vdudy         += norm * P_v * fgc / hh;
            Ix_dhdx_vdvdy    += -norm * dd * P_v * P_dudx;
            Ix_dhdx_udvdx    += norm * dd * fg * P_dvdx;
            Ix_dhdx_d2vdx2   += norm * dd * (c[g] * (3 * dd * fxxg - 3 * c[g]
                                * dd * x1 * fxgc + 3 * d2 * fxg - 3 * c[g] * d2
                                * x1 * fgc + d3 * fg + dd * x1 * x1 * c[g] * (3
                                * fgc + c[g] * fgcc)) - (hh * fxxxgg + 3 * dd
                                * fxxgg + 3 * d2 * fxgg + d3 * fgg));
            Ix_d2u_dx2       += norm * ((fxxg + c[g] * (-2 * fxgc * x1 + (c[g]
                                * fgcc * x1 + 2 * fgc) * x1 * x1) - fgc * d2 /
hh));
            Ix_d2u_dy2       += norm * fgcc / hh / hh;
            Ix_dhdx_d2vdy2   += norm * dd * (-fxgc + (fgc + fgcc * c[g]) * x1)
                                / hh;
            Ix_dudt          -= norm * c[g] * P_dhdt * fgc / hh;
            Ix_dhdx_dvdt     += norm * dd * ((c[g] * (-c[g] * x1 * fgc + fxg)
                                - fxgg) * P_dhdt + (-fgg + c[g] * fg) *
                                P_d2hdxdt);
            if(j==0) q=2;
        }

        for(i=0;i<nic;i++){
            ik = i * nkc - 1;
            for(ssum1=ssum2=ssum3=ssum4=0,k=0;k<nkc;k++){
                    ik++;
                    ssum1 += d[ik] * fc0(k,xi);
                    ssum2 += d[ik] * fc1(k,xi);
                    ssum3 += d[ik] * fc2(k,xi);
                    ssum4 += d[ik] * fc3(k,xi);
            }
            sums1[i] = ssum1;
            sums2[i] = ssum2;
            sums3[i] = ssum3;
            sums4[i] = ssum4;
        }
        cvc = cv;
        qc=5;
        for(j=0;j<=c2;j++){
            qc=6-qc;
            if(j==c2) qc=1;
            nor=hh*qc*delc/3;
            fg = fxg = fxxg = fgc = fgg = fxgg = fxxgg = fgg1 = fxgg1 = fxgc
=
            fgcc = fxxgg1 = fxxxgg = 0;
            for(i=0;i<nic;i++){
                hp0      = gc0(i,cvc);
                hp1      = gc1(i,cvc);
                hp2      = gc2(i,cvc);
                hp3      = gc3(i,cvc);
                hp4      = gg(i,cvc);
                hp5      = gg(i,1);
                fg      += sums1[i] * hp0;
                fxg     += sums2[i] * hp0;
                fxgc    += sums2[i] * hp1;
            fxxg    += sums3[i] * hp0;
            fgc     += sums1[i] * hp1;
            fgcc    += sums1[i] * hp2;
            fgg     += sums1[i] * hp4;
            fxgg    += sums2[i] * hp4;
            fxxgg   += sums3[i] * hp4;
            fxxxgg  += sums4[i] * hp4;
            fgg1    += sums1[i] * hp5;
            fxgg1   += sums2[i] * hp5;
            fxxgg1  += sums3[i] * hp5;
            ii1(ff,i) += nor * cvc * hp0;
            ii2(ff,i) += nor * hp4;
        }
        P_dvdx          = cvc * (2 * dd * fxg + d2 * fg - cvc * fgc *
                          dd * x1) -(hh * fxxgg + 2 * dd * fxgg + d2 * fgg)
;
        Iy_udvdx[ff] += nor * fg * P_dvdx;
        P_dhdt          = -hh * fxgg1 - dd * fgg1;
        P_d2hdxdt         = -2 * dd * fxxgg1 - hh * fxxxgg1 - d2 * fgg1;
        P_v               = cvc * fg * dd - hh * fxgg - dd * fgg;
        Iy_dvdt[ff]  += nor * ((cvc * (-cvc * x1 * fgc + fxg)
                          - fxgg) * P_dhdt + (-fgg + cvc * fg) *
                          P_d2hdxdt);
        Iy_d2vdx2[ff]+= nor * (cvc * (3 * dd * fxxg - 3 * cvc
                          * dd * x1 * fxgc + 3 * d2 * fxg - 3 * cvc * d2
                          * x1 * fgc + d3 * fg + dd * x1 * x1 * cvc * (3
                          * fgc + cvc * fgcc)) - (hh * fxxxgg + 3 * dd
                          * fxxgg + 3 * d2 * fxgg + d3 * fgg));
        cvc -= delc;
        if(j==0) qc=2;
    }
    // ***  p1-p0  ***

    p0_p1 = prss[ff];
    p4_p0 = -prss[nk];

    if(ff==0){
      p4_p1 = p4_p0 + p0_p1;}
    else
        p4_p1 = -prss[ff-1] + p0_p1;

    p2_p3 = Ix_d2u_dy2 + sqraver * Ix_d2u_dx2 - re * aver * (Ix_dudt +
            Ix_ududx + Ix_vdudy) + c[g] * sqraver * ( Ix_dhdx_d2vdy2 +
            sqraver * Ix_dhdx_d2vdx2 - re * aver * (Ix_dhdx_dvdt +
            Ix_dhdx_udvdx + Ix_dhdx_vdvdy));

    //--- calculation of p2-p1 ---
    Iy_d2v_dy2 = funct_dvdy(x[ff],1,hh,dd)
                 - funct_dvdy(x[ff],c[g],hh,dd);
    coprex= Sqr(funct_v(x[ff],1,hh,dd));
    Iy_vdvdy=funct_v(x[ff],c[g],hh,dd);
    Iy_vdvdy=0.5*(coprex-Sqr(Iy_vdvdy));

   //*** (p4-p3) ***
```

```
        if(ff==0){
          p3_p4 = p34;}
        else
            p3_p4 = p1_p2;

        //*** (p2-p1)***

        coprex = sqraver * Iy_d2v_dy2 + Sqr(sqraver) * Iy_d2vdx2[ff];
        p1_p2  = -coprex + re*sqraver*aver * (Iy_dvdt[ff] +
                 Iy_udvdx[ff] + Iy_vdvdy);

        rd = p1_p2 + p2_p3 - p3_p4 + p4_p1;

        //calculate the alfa elements
        for(k=0;k<nkc;k++){
           for(i=0;i<nic;i++){
           // Ix_dudt
              alfap_1 = re * aver * gc0(i,c[g]) * kk1[k];

           // c*Ix_dhdx_dvdt
              alfap_2 = re*sqraver*aver*c[g]*(kk2[k] * c[g] * gc0(i,c[g])
                        - (kk3[k] + kk2[k]) * gg(i,c[g]));

              if(ff==0){
           // Iy_dvdt_xis1
                alfap_4 = re*sqraver*aver*(fc0(k,1) * dhdx_xis1
                          * i11[i] -(h_xis1 * fc1(k,1) + dhdx_xis1
                          * fc0(k,1)) * i12[i]);
              }
              else alfap_4 = -alfap_3(i,k);
           // Iy_dvdt
              alfap_3(i,k) = -re*sqraver*aver*(fc0(k,x[ff]) * dhdx[ff]
                             * ii1(ff,i) -(hx[ff] * fc1(k,x[ff]) + dhdx[ff]
                             * fc0(k,x[ff])) * ii2(ff,i));

           // alfa-elements
              helpa[i * nkc + k] = alfap_1 + alfap_2 + alfap_3(i,k) + alfap_
4;
//              if(fabs(helpa[i*nkc+k])>mx) mx = fabs(helpa[i*nkc+k]);

           }
        }

        if(step==1){
          rd *= timestep;
          for(i=0;i<nic;i++)
             for(k=0;k<nkc;k++)
                rd += helpa[i*nkc+k] * d[i*nkc+k];
        }
        else{
          rd *= 1.5 * timestep;
          for(i=0;i<nic;i++)
             for(k=0;k<nkc;k++){
                ik = i * nkc + k;
                rd += helpa[ik] * (d[ik] - 0.5 * timestep * e_old[ik]);
            }
        }
        helpa[nkc*nic] = rd;

        for(k=0;k<=nkc*nic;k++)
           for(i=k;i<=nic*nkc;i++)
              ah(k,i) += helpa[k] * helpa[i];
     }    // end of ff loop
     RightWall(ah,helpa,h_xis1,c[g]);
 }        // end of g loop
}

//--------------------------- End of ContourInt ---------------------------
-

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

  ** Procedure FactorCalculation:   v, dh/dt, Prss, d2h/dxdt, d3h/dtdx2  **

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


 In this procedure the abovementioned expressions are calculated.
 The calculated expressions are used further on in the programm.
*/

void TimeStep::FactorCalculation()
{
 int f,i,k,ik;
 double  h1,h2,ndef,hp0,hp1,hp2,hp3,sqraver,sqraverdhdx;

 sqraver = aver * aver;

 for(f=0;f<nk;f++){
    v[f]=dudx[f]=d2hdxdt[f]=d3hdtdx2[f]=0;
    for(k=0;k<nkc;k++){
       hp0  = fc0(k,x[f]);
       hp1  = fc1(k,x[f]);
       hp2  = fc2(k,x[f]);
       hp3  = fc3(k,x[f]);
       for(i=0;i<nic;i++){
          ik           = i * nkc +k;
          v[f]         += d[ik] * (gc0(i,1) * dhdx[f] * hp0
                             - (hx[f] * hp1 + dhdx[f] * hp0) * gg(i,1));
          dudx[f]      += d[ik] * (hp1 * gc0(i,1) - gc1(i,1) * hp0 * dhdx[f]
                             / hx[f]);
          d2hdxdt[f]   += d[ik] * (- d2hdx2[f] * hp0 - 2.0 * dhdx[f] * hp1
                             - hx[f] * hp2) * gg(i,1);
          d3hdtdx2[f] += d[ik] * (-hp3 * hx[f] - 3.0 * hp1 * d2hdx2[f]
                             - 3.0 * hp2 * dhdx[f] - hp0 * d3hdx3[f]) * gg(i,1)
;
       }
    }
    dhdt[f]       = v[f] - uc[f] * dhdx[f];
    sqraverdhdx = sqraver * Sqr(dhdx[f]) ;
    ndef          = 1 + sqraverdhdx;
    prss[f]       = -sqraver * (sgr[f] * d2hdx2[f] / (ndef * sqrt(ndef)) + 2
                    * ndef * dudx[f] / (2 - ndef) + 2 * sx[f] * dhdx[f]
                    / (2 - ndef));
 }
 h1            = funct_h(1);
 h2            = funct_dhdx(1);
 sqraverdhdx = sqraver * h2 * h2;
 ndef          = 1 + sqraverdhdx;
 prss[nk]      = -sqraver * (sgr[nk] * funct_d2hdx2(1) / (ndef * sqrt(ndef))
- 2
                 * ndef * funct_dvdy(1,1,h1,h2) / (2 - ndef) + 2 * sx[nk] *
                 h2 / (2 - ndef));
}
//----------------------- End of FactorCalculation ------------------------
-

/**************************************************************************
*
 ***           A  T i m e S t e p                                        **
*
 **************************************************************************
*/

void TimeStep::ATimeStep()
{
 int i;
double xx;

 FactorCalculation();
 matrix ah(nkc*nic+1,nkc*nic+1);
 ContourInt(ah);
 cout << xy(10,9);
 Solve_D_WithContourInt(ah);
 tmplstmstp();
 NewH();
 NewNkNi_uc_h();
 SxSgrComputation();
 Solve_D_WithTangTens_Cond(ah);
 NewUH();

for(xx=0,i=0;i<nkc*nic;i++) xx += d[i]*d[i]; xx = sqrt(xx);
 ofstream output("outdata", ios::app);
 output << xx << "time= " << setprecision(2) << time << " d=" << setprecisi
on(10)
        << setiosflags(ios::showpoint | ios::fixed);
 for(i=0;i<nkc*nic;i++)
    output << setw(15) <<d[i];
 output << endl;
 output << setw(11) << " " << "b=";
 for(i=0;i<=Ncoef_b;i++)
    output << setw(15) << b[i];
 output << endl;
 resetiosflags(ios::showpoint |ios::fixed);
}

//---------------------- End of ATimeStep ------------------------------
```

```
#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <math.h>
#include "Definitions.hpp"
#include "MaraVar.hpp"
#include "Basics.hpp"
#include "RunScr.hpp"
#include "c_Computation.hpp"
#include "Init_New_Run.hpp"
#include "EnterData.hpp"
#include "Data.hpp"
#include "hGradient.hpp"
#include "NewNkNi.hpp"
#include "TimeStep.hpp"
#include "SiVaD.hpp"
#include "SolveWTT_Cond.hpp"
#include "New_DB.hpp"
#include "intvec.hpp"
#include "gvector.hpp"
#include "matrix.hpp"

void SiVaD::Reconstruct2(int n,matrix & vv)
{
 int i,j,max;
 double hlp1,sup;
 intvec ib(n);

 for(i=0;i<n;i++) ib[i]=0;
 for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
       if(helpa[i]>helpa[j]){ib[i]+=1;} else ib[j]+=1;

 max = 0;
 while(ib[max] != n-1) max += 1;
 sup = 100.0 / helpa[max];

 for(i=0;i<n;i++) helpa[i] *= sup;

 for(i=0;i<n;i++) rightvec[i] = 0.0;

 for(i=0;i<n;i++)
    if(helpa[i] > 0.01){
      for(hlp1=0.0,j=0;j<n;j++)
         hlp1 += bvec[j] * vv(j,i);
      hlp1 /= (helpa[i] / sup);
      for(j=0;j<n;j++)
         rightvec[j] += vv(j,i) * hlp1;
    }

 for(i=0;i<n;i++) bvec[i] = rightvec[i];
}
```

```
void SiVaD::Reconstruction(int n,int l,matrix & ah,matrix & vv)
{
 int i,j,k,max;
 intvec ib(n+1);
 double hlp1,sup,r;

 for(i=0;i<n;i++) ib[i]=0;
 for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
       if(helpa[i]>helpa[j]){ib[i]+=1;} else ib[j]+=1;

 max = 0;
 while(ib[max] != n-1) max += 1;
 sup = 100.0 / helpa[max];

 for(i=0;i<n;i++) helpa[i] *= sup;

 for(i=0;i<n-1;i++)
    for(j=0;j<n-1;j++)
       ah(i,j)=0.0;
 for(i=0;i<n-1;i++) bvec[i]=0.0;

 for(i=0;i<n;i++)
    if(ib[i]>=n-1){
      hlp1 = helpa[i] * vv(n-1,i);
      for(j=0;j<n-1;j++){
         bvec[j] += hlp1*vv(j,i);
         for(k=0;k<n-1;k++)
            ah(j,k) += helpa[i] * vv(j,i) * vv(k,i);
      }
    }
}

void SiVaD::Jacobi(int n,matrix & ah,matrix & vv)
{
 int j,iq,ip,i;
 double tresh,theta,tau,t,sm,s,h,g,c;
 vector bv(n+1),z(n+1);

 for(ip=0;ip<n;ip++){
    for(iq=0;iq<n;iq++) vv(ip,iq)=0.0;
       vv(ip,ip)=1.0;
 }
 for(ip=0;ip<n;ip++){
    bv[ip]=helpa[ip]=ah(ip,ip);
    z[ip]=0.0;
 }
 for(i=0;i<50;i++){
    sm=0.0;
    for(ip=0;ip<n-1;ip++){
       for(iq=ip+1;iq<n;iq++)
          sm += fabs(ah(ip,iq));
    }
    if(sm==0.0) return;
    if(i<3)
      tresh=0.2*sm/(n*n);
    else
     tresh=0.0;
    for(ip=0;ip<n-1;ip++){
       for(iq=ip+1;iq<n;iq++){
          g=100.0*fabs(ah(ip,iq));
          if(i>3 && (fabs(helpa[ip])+g)==fabs(helpa[ip])
             && (fabs(helpa[iq])+g) == fabs(helpa[iq]))
             ah(ip,iq)=0.0;
          else if(fabs(ah(ip,iq))>tresh){
           h=helpa[iq]-helpa[ip];
           if((fabs(h)+g)==fabs(h))
             t=(ah(ip,iq))/h;
           else{
                theta=0.5*h/(ah(ip,iq));
                t=1.0/(fabs(theta)+sqrt(1.0+theta*theta));
                if(theta<0.0) t = -t;
           }
           c=1.0/sqrt(1+t*t);
           s=t*c;
           tau=s/(1.0+c);
           h=t*ah(ip,iq);
           z[ip] -= h;
           z[iq] += h;
           helpa[ip] -= h;
           helpa[iq] += h;
           ah(ip,iq)=0.0;
           for(j=0;j<=ip-1;j++){
              g=ah(j,ip);h=ah(j,iq);ah(j,ip)=g-s*(h+g*tau);
              ah(j,iq)=h+s*(g-h*tau);
           }
           for(j=ip+1;j<=iq-1;j++){
              g=ah(ip,j);h=ah(j,iq);ah(ip,j)=g-s*(h+g*tau);
              ah(j,iq)=h+s*(g-h*tau);
           }
           for(j=iq+1;j<n;j++){
              g=ah(ip,j);h=ah(iq,j);ah(ip,j)=g-s*(h+g*tau);
              ah(iq,j)=h+s*(g-h*tau);
           }
           for(j=0;j<n;j++){
              g=vv(j,ip);h=vv(j,iq);vv(j,ip)=g-s*(h+g*tau);
              vv(j,iq)=h+s*(g-h*tau);
           }
         }
      }
   }
   for(ip=0;ip<n;ip++){
      bv[ip] += z[ip];
      helpa[ip]=bv[ip];
      z[ip]=0.0;
   }
}
cout << "Too many iterations in routine JACOBI" << endl;
```

```
}

void SiVaD::Simq(vector & bv,int n,matrix & ah)
{
 /* Purpose:
        Obtain solution of the linear equation ah(transpose)x = b.
        Note that this routine solves the equation ah x = b only in
        case matrix ah is symmetric.
    Description of variables:
        ah - Matrix of coefficients.
             The size of the matrix is n x n.
             The coefficients are of type double.
             These are destroyed in the computation.
        b  - Vector of original constants (length n).
             These are replaced by the final solution values, vector x.
        n  - Number of equations and variables. n must be greater than 1.
    Derived from fortran source in I.B.M. system/360 scientific subroutine
        package, version III.
 */

 double tol,biga,save;
 int jj,j,jy,i,imax,i1,k,ix,jx,ny,ib,ic;
 tol = 0;
 jj = -n;
 for(j=0;j<n;j++){
    jy = j + 2;
    jj += n + 1;
    biga = 0;
    for(i=j;i<n;i++)
       if(fabs(biga)<fabs(ah(j,i))){
         biga = ah(j,i);
         imax = i;
       };
    if(fabs(biga)<=tol) cerr << "Matrix a is singular\n";
    i1 = j + 1 + n * (j-1);
    for(k=j;k<n;k++){
       i1 += n;
       save = ah(k,j);
       ah(k,j) = ah(k,imax);
       ah(k,imax) = save;
       ah(k,j) /= biga;
    };
    save = bv[imax];
    bv[imax] = bv[j];
    bv[j] = save / biga;
    if(j!=n-1)
      for(ix=jy-1;ix<n;ix++){
         for(jx=jy-1;jx<n;jx++){
            ah(jx,ix) -= ah(j,ix) * ah(jx,j);
         };
         bv[ix] -= bv[j] * ah(j,ix);
      };
 };
 ny = n - 1;
 for(j=0;j<ny;j++){
    ib = n - j - 2;
    ic = n - 1;
    for(k=1;k<=j+1;k++){
       bv[ib] -= ah(n-k,n-j-2) * bv[ic];
       ic--;
    }
 }
}

void SolveWTT_Cond::MatSolveWithRightWall(int n,matrix & ah)
{
 int f,k;

 Simq(bvec,n,ah);
 for(f=0;f<n;f++)
    for(k=0;k<n;k++)
       ah(f,k) = 0.0;
}

//---------------------- End of ATimeStep ------------------------------

/***********************************************************************
*
 ***      S O L V E _ D _ W I T H T A N G T E N S _ C O N D          **
*
 ***********************************************************************
*
*/
void SolveWTT_Cond::Solve_D_WithTangTens_Cond(matrix & ah)
{
 int f,i,k,n;
 double h1;

 n = nkc * nic + 1;
 for(i=0;i<n;i++)
    for(k=0;k<n;k++)
       ah(i,k) = 0.0;

 for(f=0;f<nk;f++){
    for(i=0;i<nic;i++){
       for(k=0;k<nkc;k++){
          h1 = (1 - aver * aver * dhdx[f] * dhdx[f]) * (fc0(k,x[f]) * gc1(i
,1)
                 / hx[f] + aver * aver * (fc1(k,x[f]) * 2 * gc0(i,1) * dhdx[f
]
                 + fc0(k,x[f]) * gc0(i,1) * d2hdx2[f] - fc0(k,x[f]) * gc1(i,1
)
                 * dhdx[f] * dhdx[f] / hx[f] - (fc2(k,x[f]) * hx[f]
                 + fc1(k,x[f]) * 2 * dhdx[f] / hx[f] + fc0(k,x[f]) * d2hdx2[f
]
                 / hx[f]) * gg(i,1))) + 4 * aver * aver * dhdx[f] * (fc0(k,x[
f])
                 * gc1(i,1) * dhdx[f] / hx[f] - fc1(k,x[f]) * gc0(i,1));
          helpa[i*nkc+k] = h1;
       }
    }

    helpa[n-1] = (1 + aver * aver * dhdx[f] * dhdx[f]) * sx[f];

    for(i=0;i<n;i++)
       for(k=0;k<n;k++)
          ah(i,k) += helpa[i] * helpa[k];
 }

 matrix vv(nkc*nic+1,nkc*nic+1);
 Jacobi(n,ah,vv);
 Reconstruction(n,nkc,ah,vv);

 for(i=0;i<n-1;i++){
    bvec[i] += rightvec[i];
    for(k=0;k<=i;k++){
       ah(i,k) += leftmat[i*(i+1)/2+k];
       ah(k,i) = ah(i,k);
    }
 }

 Jacobi(n-1,ah,vv);
 Reconstruct2(n-1,vv);

 for(i=0;i<nic;i++)
    for(k=0;k<nkc;k++){
       d[i*nkc+k] = bvec[i*nkc+k];
       bvec[i*nkc+k] = 0;
    }
}

/**************************************************************************
*
 ***      S O L V E _ D _ W I T H C O N T O U R I N T                    **
*
 *************************************************************************
*
*/
void SolveWTT_Cond::Solve_D_WithContourInt(matrix & ah)
{
/*Here is calculated the time derivative of the boundery condition of the
  tangential stress condition at the interface.
  This supplies a contribution to the matrix ah and the vector bvec for the
  computation of the time derivative of the d values, i.e. the e values.
  Left out for the moment.
*/

 int k,i,n;

 n = nic * nkc + 1;

for(k=1;k<n;k++)
```

```
    for(i=0;i<k;i++)
       ah(k,i) = ah(i,k);

  matrix vv(nkc*nic+1,nkc*nic+1);
  Jacobi(n,ah,vv);
  Reconstruction(n,n-nkc-1,ah,vv);

  for(i=0;i<n-1;i++){
     rightvec[i] = bvec[i];
     for(k=0;k<=i;k++)
        leftmat[i*(i+1)/2+k] = ah(i,k);
  }
}
//------------------- End of Solve_D_WithContourInt-----------------------


/***********************************************************************
*
 ***    N E W U H                                                      **
*
 **********************************************************************
*

  Compute delu/delt, du/dy, du/dt, Integrate du/dt en dh/dt
  The change in the horizontal velocity u and the height h over a
  time step timestep can be computed with the help of the e factors
  descended from the contour integrals.
*/

void TimeStep:: NewUH()
{
 int i;

 if(par!=2 && step==1){
   for(i=0;i<nic*nkc;i++){
      e_old[i] = (d[i] - d_old[i]) / timestep;
      d_old[i] = d[i];
   }
 }
 else{
   for(i=0;i<nic*nkc;i++){
      e_old[i] = e_old[i]/3 + 2.0 * (d[i] - d_old[i]) / (3.0 * timestep);
      d_old[i] = d[i];
   }
 }
}

//----------------------- End of New UH ---------------------------------
```

```
#include "Definitions.hpp"
#include "vector.hpp"
vector d(nkcmax*nicmax),e_old(nkcmax*nicmax),d_old(nkcmax*nicmax),
       bt_old(10*ncoefmax),bt(10*ncoefmax),
       leftmat(nkcmax*nicmax*(nkcmax*nicmax+1)/2),
       helpa(nkcmax*nicmax+1),bvec(nkcmax*nicmax),rightvec(nkcmax*nicmax);
```

```
#include <iostream.h>
#include <stdlib.h>
#include "matrix.hpp"
#include "vector.hpp"

void matrix::errors(){
  cerr << "There is not enough memory" << endl;
  exit(1);
}

matrix & matrix::operator = (const matrix & ha){
 if(ha.rows() != r || ha.cols() != c){
   // different dimensions, re-allocate
   deletematrix();
   r = ha.rows(); c = ha.cols();
   newmatrix();
 }
 for(unsigned i=0;i<r;i++)
    for(unsigned j=0;j<c;j++)
       base[i][j] = ha(i,j);
 return *this;
}

void matrix::newmatrix(){
 if ((base = new double * [r]) == NULL) errors();
 for (unsigned i=0;i<r;i++)
   if ((base[i] = new double [c]) == NULL) errors();
 unsigned j,k;
 for (j=0;j<r;j++) for(k=0;k<c;k++) base[j][k] = 0.0;
}

void matrix::deletematrix(){
 if(base){
   for(unsigned i=0;i<rows();i++) if(base[i]) delete[] base[i];
   delete[] base;
 }
}

double & matrix::operator () (unsigned i, unsigned j) const{
 if(i>=rows()){
     cerr << "matrix: first index too large!" << endl;
     exit(1);
   }
   if(j>=cols()){
     cerr << "matrix: second index too large!" << endl;
     exit(1);
   }
   return base[i][j];
}

void vector::new_vector(){
  if((base=new double [dimension])==NULL) errors();
  for(unsigned i=0;i<dimension;i++) base[i] = 0.0;
}
```

```
double & vector::operator [] (unsigned i) const {
  if(i>=dim()){ cerr << "vector: index too large!" << endl;
     exit(1);
  }
  return base[i];
}
```

```
#include <iostream.h>
#include <math.h>
#include "Definitions.hpp"
#include "Data.hpp"

double Data::gc0(int i, double c)
{
 double g0;
 switch (i){
   case 0   : g0 = c * exp(-100.0*(1-c)); break;
   case 1   : g0 = c * exp(-10.0*(1-c));  break;
   case 2   : g0 = c * exp(-3.0*(1-c));  break;
   default : g0 = pow(c,i-2);
 }
 return g0;
}

double Data::gc1(int i, double c)
{
 double g1;
 switch (i){
   case 0   : g1 = (100 * c + 1) * exp(-100.0*(1-c)); break;
   case 1   : g1 = (10 * c + 1) * exp(-10.0*(1-c));   break;
   case 2   : g1 = (3 * c + 1) * exp(-3.0*(1-c));   break;
   default : g1 = (i-2) * pow(c,i-3);
 }
 return g1;
}

double Data::gc2(int i, double c)
{
 double g2;
 switch (i){
   case 0   : g2 = 100 * (100 * c + 2) * exp(-100.0*(1-c)); break;
   case 1   : g2 = 10 * (10 * c + 2) * exp(-10.0*(1-c));    break;
   case 2   : g2 = 3 * (3 * c + 2) * exp(-3.0*(1-c));    break;
   default : g2 = (i-2) * (i-3) * pow(c,i-4);
 }
 return g2;
}

double Data::gc3(int i, double c)
{
 double g3;
 switch (i){
   case 0   : g3 = 100 * 100 * (100 * c + 3) * exp(-100.0*(1-c)); break;
   case 1   : g3 = 10 * 10 * (10 * c + 3) * exp(-10.0*(1-c));     break;
   case 2   : g3 = 3 * 3 * (3 * c + 3) * exp(-3.0*(1-c));     break;
   default : g3 = (i-2) * (i-3) * (i-4) * pow(c,i-5);
 }
 return g3;
}

double Data::gg(int i, double c)
```

```
{
 double ge;
 switch (i){
   case 0   : ge = 0.01 * (c - 0.01) * exp(-100.0*(1-c)); break;
   case 1   : ge = (c - 1.0/10) * exp(-10.0*(1-c)) / 10;  break;
   case 2   : ge = (c - 1.0/3) * exp(-3.0*(1-c)) / 3;  break;
   default : ge = pow(c,i-1) / (i-1);
 }
 return ge;
}
```

```
#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <math.h>
#include <fstream.h>
#include "Definitions.hpp"
#include "MaraVar.hpp"
#include "Basics.hpp"
#include "hGradient.hpp"
#include "Function.hpp"
#include "New_DB.hpp"
#include "TimeStep.hpp"
#include "gvector.hpp"

//*************class Data : public virtual MaraVar *******************

inline double Cube(double r){ return r*r*r; }

void Data::b_init()
{
 b[0] = 1;
}

void TimeStep::RightWall(matrix & ah,vector & helpa,double h_xis1,double c)
{
int i,k;
double rd;
    for(i=0;i<nic;i++){
       for(k=0;k<nkc;k++)
          helpa[i*nkc+k] = (-pi * (k+1) * OddNeg(k+1)) * h_xis1 * gg(i,c);
    }

    if(step==1){
      for(rd=0,i=0;i<nic;i++)
         for(k=0;k<nkc;k++)
            rd += helpa[i*nkc+k] * d[i*nkc+k];
    }
    else{
      for(rd=0,i=0;i<nic;i++)
         for(k=0;k<nkc;k++)
            rd += helpa[i*nkc+k] * (d[i*nkc+k] - 0.5 * timestep * e_old[i*n
kc+k]);
    }
    helpa[nkc*nic] = rd;

    for(k=0;k<=nkc*nic;k++)
       for(i=k;i<=nic*nkc;i++)
          ah(k,i) += helpa[k] * helpa[i];
}

double Data::fc0(int k, double x)
{
 return sin(pi*(k+1)*x);
}

double Data::fc1(int k, double x)
{
 return pi*(k+1)*cos(pi*(k+1)*x);
}

double Data::fc2(int k, double x)
{
 return -pi*pi*(k+1)*(k+1)*sin(pi*(k+1)*x);
}

double Data::fc3(int k, double x)
{
 return -Cube(pi*(k+1))*cos(pi*(k+1)*x);
}

//*******class hGradient : public virtual Functions*************

/**********************************************************************
  **                  Procedure hGradientsCalculation:              **
  **       Computation of the derivatives of h with regard to x     **
  ********************************************************************
*/



void hGradient::hGradientCalculation()
{
 for(int f=0;f<nk;f++){
       hx[f] = funct_h(x[f]);
       dhdx[f] = funct_dhdx(x[f]);
       d2hdx2[f] = funct_d2hdx2(x[f]);
       d3hdx3[f] = funct_d3hdx3(x[f]);
 }
}

//-------------------- End of hGradientsCalculation ----------------------

//*********class Functions : public virtual Data*********

/**********************************************************************
  ** Function h(Xi)                                                 **
  ********************************************************************
*/

double Functions::funct_h(double xi)
{
 double ha=1.0;
 for(int i=1;i<=Ncoef_b;i++) ha += b[i] * cos(pi*i*xi);
 return ha;
}
//---------------------- End of funct_h-----------------------------------
```

```
/***********************************************************************
  ** Function dhdx(Xi)                                                **
  **********************************************************************
*/

double Functions::funct_dhdx(double xi)
{
 double dhadx=0.0;
 for(int i=1;i<=Ncoef_b;i++) dhadx -= i * pi * b[i] * sin(pi*i*xi);
 return dhadx;
}
//---------------------- End of funct_dhdx--------------------------------

/***********************************************************************
  ** Function d2hdx2(Xi)                                              **
  **********************************************************************
*/

double Functions::funct_d2hdx2(double xi)
{
 double d2hadx2=0.0;
 for(int i=1;i<=Ncoef_b;i++) d2hadx2 -= i * i * pi * pi * b[i] * cos(pi*i*x
i);
 return d2hadx2;
}
//---------------------- End of funct_d2hdx2------------------------------
-

/***********************************************************************
  ** Function d3hdx3(Xi)                                              **
  **********************************************************************
*/

double Functions::funct_d3hdx3(double xi)
{
 double d3hadx3=0.0;
 for(int i=1;i<=Ncoef_b;i++) d3hadx3 += Cube(i*pi) * b[i] * sin(pi*i*xi);
 return d3hadx3;

}
//---------------------- End of funct_d3hdx3------------------------------
-

void New_DB::NewH()
{
 int i,j,k,mx,z2,q;
 double del,xv,nor,hc1,ef,ha,dhadx,dhadt;

 mx = Ncoef_b;
 if(Ncoef_b<nic) mx = nic;
 if(mx<nkc) mx = nkc;
 vector cosk(mx+1),sink(mx+1),bnn(Ncoef_b+1);

 z2  = 512;
```

```
  q   = 5;
  del = 1.0/z2;
  xv  = -del;
  for(j=0;j<=z2;j++){
     q=6-q;
     if(j==z2) q=1;
     nor = q * del / 3;
     xv += del;

     cosk[0] = 1; sink[0] = 0;
     cosk[1] = cos(pi*xv); sink[1] = sin(pi*xv);
     hc1 = 2 * cosk[1];
     for(i=2;i<=mx;i++){
        cosk[i] = hc1 * cosk[i-1] - cosk[i-2];
        sink[i] = hc1 * sink[i-1] - sink[i-2];
     }

     for(ha=1.0,i=1;i<=Ncoef_b;i++) ha += b[i] * cosk[i];
     for(dhadx=0.0,i=1;i<=Ncoef_b;i++) dhadx -= pi * i * b[i] * sink[i];

     for(dhadt=0.0,k=0;k<nkc;k++) for(i=0;i<nic;i++)
        dhadt -= d[i*nkc+k] * (sink[k+1] * dhadx + pi * (k+1) * cosk[k+1] *
ha)
                 * gg(i,1);

     ef = ha - 1.0 + dhadt * timestep;

     for(i=1;i<=Ncoef_b;i++) bnn[i] += 2 * nor * ef * cosk[i];
     if(j==0) q=2;
  }

  for(i=1;i<=Ncoef_b;i++) b[i] = bnn[i];
}
```

```
#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <math.h>
#include <fstream.h>
#include "Definitions.hpp"
#include "MaraVar.hpp"
#include "Basics.hpp"
#include "hGradient.hpp"
#include "Function.hpp"
#include "New_DB.hpp"
#include "TimeStep.hpp"
#include "gvector.hpp"

//*************class Data : public virtual MaraVar ******************

void Data::b_init()
{
 b[0] = 1;
}

void TimeStep::RightWall(matrix & ah,vector & helpa,double h_xis1,double c)
{}

double Data::fc0(int k, double x)
{
 double mu;

 switch (k){
   case 0 :  mu = 3.9266023120479187783; break;
   case 1 :  mu = 7.0685827456287320886; break;
   case 2 :  mu = 10.210176122813030546; break;
   case 3 :  mu = 13.351768777754093124; break;
   default : mu = (k + 1.25) * pi;
 }
 return sinh(mu*x)/sinh(mu) - sin(mu*x)/sin(mu);
}

double Data::fc1(int k, double x)
{
 double mu;

 switch (k){
   case 0 :  mu = 3.9266023120479187783; break;
   case 1 :  mu = 7.0685827456287320886; break;
   case 2 :  mu = 10.210176122813030546; break;
   case 3 :  mu = 13.351768777754093124; break;
   default : mu = (k + 1.25) * pi;
 }
 return mu * (cosh(mu*x)/sinh(mu) - cos(mu*x)/sin(mu));
}

double Data::fc2(int k, double x)
```

```
{
 double mu;

 switch (k){
   case 0 :  mu = 3.9266023120479187783; break;
   case 1 :  mu = 7.0685827456287320886; break;
   case 2 :  mu = 10.210176122813030546; break;
   case 3 :  mu = 13.351768777754093124; break;
   default : mu = (k + 1.25) * pi;
 }
 return mu * mu * (sinh(mu*x)/sinh(mu) + sin(mu*x)/sin(mu));
}

double Data::fc3(int k, double x)
{
 double mu;

 switch (k){
   case 0 :  mu = 3.9266023120479187783; break;
   case 1 :  mu = 7.0685827456287320886; break;
   case 2 :  mu = 10.210176122813030546; break;
   case 3 :  mu = 13.351768777754093124; break;
   default : mu = (k + 1.25) * pi;
 }
 return mu * mu * mu * (cosh(mu*x)/sinh(mu) + cos(mu*x)/sin(mu));

}

//*******class hGradient : public virtual Functions*************

/***************************************************************************
  **                   Procedure hGradientsCalculation:                    **
  **       Computation of the derivatives of h with regard to x            **
  *************************************************************************
*/

inline double Cube(double r){ return r*r*r; }

void hGradient::hGradientCalculation()
{
 for(int f=0;f<nk;f++){
       hx[f] = funct_h(x[f]);
       dhdx[f] = funct_dhdx(x[f]);
       d2hdx2[f] = funct_d2hdx2(x[f]);
       d3hdx3[f] = funct_d3hdx3(x[f]);
 }
}

//-------------------- End of hGradientsCalculation ----------------------

//*********class Functions : public virtual Data********

/***************************************************************************
  ** Function h(Xi)                                                        **
  **************************************************************************
*/

double Functions::funct_h(double xi)
{
 int i;
 double ha;

 ha = 1 +
      b[1] * (cosh(2.365020372431352013O*xi)/cosh(2.3650203724313520130) -
      cos(2.3650203724313520130*xi)/cos(2.3650203724313520130)) +
      b[2] * (cosh(5.4978039190008354533*xi)/cosh(5.4978039190008354533) -
      cos(5.4978039190008354533*xi)/cos(5.4978039190008354533)) +
      b[3] * (cosh(8.6393798286997407190*xi)/cosh(8.6393798286997407190) -
      cos(8.6393798286997407190*xi)/cos(8.6393798286997407190)) +
      b[4] * (cosh(11.780972451020227538*xi)/cosh(11.780972451020227538) -
      cos(11.780972451020227538*xi)/cos(11.780972451020227538));
 for(i=5;i<=Ncoef_b;i++) ha += b[i] * (cosh((i-0.25)*pi*xi)/cosh((i-0.25)*p
i) -
                                  cos((i-0.25)*pi*xi)/cos((i-0.25)*pi));
 return ha;
}
//---------------------- End of funct_h------------------------------------

/**************************************************************************
  ** Function dhdx(Xi)                                                   **
  **************************************************************************
*/

double Functions::funct_dhdx(double xi)
{
 int i;
 double dhadx;

 dhadx = b[1] * 2.3650203724313520130 * (sinh(2.3650203724313520130*xi)
      /cosh(2.3650203724313520130) +
      sin(2.3650203724313520130*xi)/cos(2.3650203724313520130)) +
      b[2] * 5.4978039190008354533 * (sinh(5.4978039190008354533*xi)
      /cosh(5.4978039190008354533) +
      sin(5.4978039190008354533*xi)/cos(5.4978039190008354533)) +
      b[3] * 8.6393798286997407190 * (sinh(8.6393798286997407190*xi)
      /cosh(8.6393798286997407190) +
      sin(8.6393798286997407190*xi)/cos(8.6393798286997407190)) +
      b[4] * 11.780972451020227538 * (sinh(11.780972451020227538*xi)
      /cosh(11.780972451020227538) +
      sin(11.780972451020227538*xi)/cos(11.780972451020227538));
 for(i=5;i<=Ncoef_b;i++) dhadx += b[i] * (i-0.25) * pi * (sinh((i-0.25)*pi*
xi)
                                  /cosh((i-0.25)*pi) +
                                  sin((i-0.25)*pi*xi)/cos((i-0.25)*pi));
 return dhadx;
}
//---------------------- End of funct_dhdx---------------------------------
```

```
/*************************************************************************
  ** Function d2hdx2(Xi)                                                **
  ************************************************************************
*/

double Functions::funct_d2hdx2(double xi)
{
 int i;
 double d2hadx2;

 d2hadx2 = b[1] * Sqr(2.3650203724313520130) * (cosh(2.3650203724313520130*
xi)
       /cosh(2.3650203724313520130) +
       cos(2.3650203724313520130*xi)/cos(2.3650203724313520130)) +
       b[2] * Sqr(5.4978039190008354533) * (cosh(5.4978039190008354533*xi)
       /cosh(5.4978039190008354533) +
       cos(5.4978039190008354533*xi)/cos(5.4978039190008354533)) +
       b[3] * Sqr(8.6393798286997407190) * (cosh(8.6393798286997407190*xi)
       /cosh(8.6393798286997407190) +
       cos(8.6393798286997407190*xi)/cos(8.6393798286997407190)) +
       b[4] * Sqr(11.780972451020227538) * (cosh(11.780972451020227538*xi)
       /cosh(11.780972451020227538) +
       cos(11.780972451020227538*xi)/cos(11.780972451020227538));
 for(i=5;i<=Ncoef_b;i++) d2hadx2 += b[i] * Sqr((i-0.25)*pi) * (cosh((i-0.25
)*pi*xi)
                                   /cosh((i-0.25)*pi) +
                                   cos((i-0.25)*pi*xi)/cos((i-0.25)*pi));
 return d2hadx2;
}
//---------------------- End of funct_d2hdx2-----------------------------
-

/*************************************************************************
  ** Function d3hdx3(Xi)                                                **
  ************************************************************************
*/

double Functions::funct_d3hdx3(double xi)
{
 int i;
 double d3hadx3;

 d3hadx3 = b[1] * Cube(2.3650203724313520130) * (sinh(2.3650203724313520130
*xi)
       /cosh(2.3650203724313520130) -
       sin(2.3650203724313520130*xi)/cos(2.3650203724313520130)) +
       b[2] * Cube(5.4978039190008354533) * (sinh(5.4978039190008354533*xi)
       /cosh(5.4978039190008354533) -
       sin(5.4978039190008354533*xi)/cos(5.4978039190008354533)) +
       b[3] * Cube(8.6393798286997407190) * (sinh(8.6393798286997407190*xi)
       /cosh(8.6393798286997407190) -
       sin(8.6393798286997407190*xi)/cos(8.6393798286997407190)) +
       b[4] * Cube(11.780972451020227538) * (sinh(11.780972451020227538*xi)
       /cosh(11.780972451020227538) -
```

```
      sin(11.780972451020227538*xi)/cos(11.780972451020227538));
 for(i=5;i<=Ncoef_b;i++) d3hadx3 += b[i] * Cube((i-0.25)*pi) *
                                     (sinh((i-0.25)*pi*xi) /cosh((i-0.25)*pi
) -
                                     sin((i-0.25)*pi*xi)/cos((i-0.25)*pi));
 return d3hadx3;

}
//----------------------- End of funct_d3hdx3------------------------------
-

void New_DB::NewH()
{
 int i,j,k,z2,q;
 double del,xv,nor,ef,ha,dhadx,dhadt,uu,la[5];

 vector csk(Ncoef_b+1),snk(Ncoef_b+1),bnn(Ncoef_b+1);

 la[1] = 2.365020372431352013O;
 la[2] = 5.497803919000835453З;
 la[3] = 8.639379828699740719O;
 la[4] = 11.780972451020227538;

 z2  = 512;
 q   = 5;
 del = 1.0/z2;
 xv  = -del;
 for(j=0;j<=z2;j++){
    q=6-q;
    if(j==z2) q=1;
    nor = q * del / 3;
    xv += del;

    csk[1] = cosh(la[1]*xv)/cosh(la[1])-cos(la[1]*xv)/cos(la[1]);
    csk[2] = cosh(la[2]*xv)/cosh(la[2])-cos(la[2]*xv)/cos(la[2]);
    csk[3] = cosh(la[3]*xv)/cosh(la[3])-cos(la[3]*xv)/cos(la[3]);
    csk[4] = cosh(la[4]*xv)/cosh(la[4])-cos(la[4]*xv)/cos(la[4]);
    for(i=5;i<=Ncoef_b;i++){
       csk[i] = cosh((i-0.25)*pi*xv)/cosh((i-0.25)*pi)-cos((i-0.25)*pi*xv)
                /cos((i-0.25)*pi);
    }
    snk[1] = la[1] * (sinh(la[1]*xv)/cosh(la[1])+sin(la[1]*xv)/cos(la[1]));
    snk[2] = la[2] * (sinh(la[2]*xv)/cosh(la[2])+sin(la[2]*xv)/cos(la[2]));
    snk[3] = la[3] * (sinh(la[3]*xv)/cosh(la[3])+sin(la[3]*xv)/cos(la[3]));
    snk[4] = la[4] * (sinh(la[4]*xv)/cosh(la[4])+sin(la[4]*xv)/cos(la[4]));
    for(i=5;i<=Ncoef_b;i++){
       snk[i] = (i-0.25) * pi * (sinh((i-0.25)*pi*xv)/cosh((i-0.25)*pi)+
                sin((i-0.25)*pi*xv) / cos((i-0.25)*pi));
    }
    for(ha=1.0,i=1;i<=Ncoef_b;i++) ha += b[i] * csk[i];
    for(dhadx=0.0,i=1;i<=Ncoef_b;i++) dhadx += b[i] * snk[i];

    for(uu=0.0,i=0;i<nic;i++) for(k=0;k<nkc;k++) uu += d[i*nkc+k] * fc0(k,x
v);
```

```
    dhadt = funct_v(xv,1,ha,dhadx) - dhadx * uu;

    ef = ha - 1.0 + dhadt * timestep;

    for(i=1;i<=Ncoef_b;i++) bnn[i] += nor * ef * csk[i];
    if(j==0) q=2;
 }

 for(i=1;i<=Ncoef_b;i++) b[i] = bnn[i];
}
```

```
#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <math.h>
#include <fstream.h>
#include "Definitions.hpp"
#include "MaraVar.hpp"
#include "Basics.hpp"
#include "hGradient.hpp"
#include "Function.hpp"
#include "New_DB.hpp"
#include "TimeStep.hpp"
#include "gvector.hpp"

//*************class Data : public virtual MaraVar *******************

inline double Cube(double r){ return r*r*r; }

void Data::b_init()
{
 b[1] = 1;
}

void TimeStep::RightWall(matrix & ah,vector & helpa,double h_xis1,double c)
{}

double Data::fc0(int k, double x)
{
 double y;
 if(k==0) y=x; else y = sin(pi*k*x);
 return y;
}

double Data::fc1(int k, double x)
{
 double dydx;
 if(k==0) dydx=1.0; else dydx = pi*k*cos(pi*k*x);
 return dydx;
}

double Data::fc2(int k, double x)
{
 double d2ydx2;
 if(k==0) d2ydx2=0.0; else d2ydx2 = -Sqr(pi*k)*sin(pi*k*x);
 return d2ydx2;
}

double Data::fc3(int k, double x)
{
 double d3ydx3;
 if(k==0) d3ydx3=0.0; else d3ydx3 = -Cube(pi*k)*cos(pi*k*x);
 return d3ydx3;
}
```

```
//*******class hGradient : public virtual Functions*************

/**************************************************************************
  **                    Procedure hGradientsCalculation:                  **
  **       Computation of the derivatives of h with regard to x           **
  **************************************************************************
*/



void hGradient::hGradientCalculation()
{
 for(int f=0;f<nk;f++){
       hx[f] = funct_h(x[f]);
       dhdx[f] = funct_dhdx(x[f]);
       d2hdx2[f] = funct_d2hdx2(x[f]);
       d3hdx3[f] = funct_d3hdx3(x[f]);
 }
}

//------------------- End of hGradientsCalculation ---------------------

//*********class Functions : public virtual Data*********

/**************************************************************************
  ** Function h(Xi)                                                       **
  **************************************************************************
*/

double Functions::funct_h(double xi)
{
 double ha=b[0]*xi*xi;
 for(int i=1;i<=Ncoef_b;i++) ha += b[i] * cos(pi*(i-1)*xi);
 return ha;
}
//---------------------- End of funct_h--------------------------------

/**************************************************************************
  ** Function dhdx(Xi)                                                    **
  **************************************************************************
*/

double Functions::funct_dhdx(double xi)
{
 double dhadx=2*b[0]*xi;
 for(int i=1;i<=Ncoef_b;i++) dhadx -= (i-1) * pi * b[i] * sin(pi*(i-1)*xi);
 return dhadx;
}
//---------------------- End of funct_dhdx--------------------------------

/**************************************************************************
  ** Function d2hdx2(Xi)                                                  **
  **************************************************************************
```

```
*/

double Functions::funct_d2hdx2(double xi)
{
 double d2hadx2=2.0*b[0];
 for(int i=1;i<=Ncoef_b;i++) d2hadx2 -= Sqr((i-1)*pi)*b[i]*cos(pi*(i-1)*xi)
;
 return d2hadx2;
}
//----------------------- End of funct_d2hdx2------------------------------
-

/**************************************************************************
  ** Function d3hdx3(Xi)                                                 **
  *************************************************************************
*/

double Functions::funct_d3hdx3(double xi)
{
 double d3hadx3=0.0;
 for(int i=1;i<=Ncoef_b;i++) d3hadx3 += Cube((i-1)*pi) * b[i] * sin(pi*(i-1
)*xi);
 return d3hadx3;

}
//----------------------- End of funct_d3hdx3------------------------------
-

void New_DB::NewH()
{
 int i,j,k,mx,z2,q;
 double del,xv,nor,hc1,ef,ha,dhadx,dhadt;

 mx = Ncoef_b;
 if(Ncoef_b<nic){ cerr << "Ncoef_b is too small"; exit(1);}
 if(mx<nkc){ cerr << "Ncoef_b is too small"; exit(1);}
 vector cosk(mx+1),sink(mx+1),bnn(Ncoef_b+1);

 z2  = 512;
 q   = 5;
 del = 1.0/z2;
 xv  = -del;
 for(j=0;j<=z2;j++){
    q=6-q;
    if(j==z2) q=1;
    nor = q * del / 3;
    xv += del;

    cosk[0] = 1; sink[0] = 0;
    cosk[1] = cos(pi*xv); sink[1] = sin(pi*xv);
    hc1 = 2 * cosk[1];
    for(i=2;i<mx;i++){
       cosk[i] = hc1 * cosk[i-1] - cosk[i-2];
       sink[i] = hc1 * sink[i-1] - sink[i-2];
```

```
     }

     for(ha=b[0]*xv*xv,i=1;i<=Ncoef_b;i++) ha += b[i] * cosk[i-1];
     for(dhadx=2*b[0]*xv,i=1;i<=Ncoef_b;i++)
        dhadx -= pi * (i-1) * b[i] * sink[i-1];

     dhadt=0.0;
     for(i=0;i<nic;i++) dhadt += d[i*nkc] * (-ha - xv * dhadx) * gg(i,1);
     for(k=1;k<nkc;k++) for(i=0;i<nic;i++)
        dhadt -= d[i*nkc+k] * (pi*k*cos(pi*k*xv) * ha + sin(pi*k*xv) * dhadx
)
                * gg(i,1);

     ef = ha + dhadt * timestep;

     bnn[0] += nor * ef * xv * xv;
     for(i=1;i<=Ncoef_b;i++) bnn[i] += nor * ef * cosk[i-1];
     if(j==0) q=2;
  }

  matrix ah(Ncoef_b+1,Ncoef_b+1);
  ah(0,0) = 0.2;
  ah(1,0) = ah(0,1) = 1.0/3;
  for(i=2;i<=Ncoef_b;i++) ah(i,0) = ah(0,i) = 2.0*OddNeg(i-1) / Sqr((i-1)*pi
);
  ah(1,1) = 1.0;
  for(i=2;i<=Ncoef_b;i++) ah(i,i) = 0.5;
  Simq(bnn,Ncoef_b+1,ah);
  for(i=0;i<=Ncoef_b;i++) b[i] = bnn[i];
}
```

```
#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <math.h>
#include <fstream.h>
#include "Definitions.hpp"
#include "MaraVar.hpp"
#include "Basics.hpp"
#include "hGradient.hpp"
#include "Function.hpp"
#include "New_DB.hpp"
#include "TimeStep.hpp"
#include "gvector.hpp"

//*************class Data : public virtual MaraVar *******************

void Data::b_init()
{
 b[0] = 1;
}

void TimeStep::RightWall(matrix & ah,vector & helpa,double h_xis1,double c)
{}

double Data::fc0(int k, double x)
{
 double k0,k1,k2;
 switch (k){
        case 0 : k2 = x; break;
        case 1 : k2 = x * (4*x*x - 3); break;
        default: k0 = x; k1 = x * (4*x*x - 3);
                 for(int i=1;i<k;i++){
                    k2 = (4*x*x - 2) * k1 - k0;
                    k0 = k1; k1 =k2;
                 } break;
        }
 return k2;
}

double Data::fc1(int k, double x)
{
 double k0,k1,k2;
 switch (k){
        case 0 : k2 = 1; break;
        case 1 : k2 = 12*x*x - 3; break;
        default: k0 = 1; k1 = 12*x*x - 3;
                 for(int i=2;i<=k;i++){
                    k2 = ( (2*i+1.0)/(2*i-1.0) ) * (4*x*x - 2) * k1
                         - ( (2*i+1.0)/(2*i-3.0) ) * k0;
                    k0 = k1; k1 =k2;
                 } break;
        }
 return k2;
```

```
}

double Data::fc2(int k, double x)
{
 double k0,k1,k2;
 switch (k){
       case 0 : k2 = 0; break;
       case 1 : k2 = 24*x; break;
       default: k0 = 0; k1 = 24*x;
                for(int i=2;i<=k;i++){
                   k2 = ( (2*i+1.0)/(2*i-1.0) ) * (4*x * ( i*x/(i-1.0) )
                        - ( (2*i*i-2*i-1.0)/(i*i-2*i+1.0) )) * k1
                        - ( (2*i+1.0)/(2*i-3.0) ) * Sqr(k/(k-1.0)) * k0;
                   k0 = k1; k1 =k2;
                } break;
       }
 return k2;
}

double Data::fc3(int k, double x)
{
 double k0,k1,k2;
 switch (k){
       case 0 : k2 = 0; break;
       case 1 : k2 = 24; break;
       default: k0 = 0; k1 = 24;
                for(int i=2;i<=k;i++){
                   k2 = ( (2*i+1.0)/(2*i-3.0) ) * (4*x * ( i*x/(i-1.0) )
                        - ( (2*i*i-2*i-3.0)/(i*i-2*i+1.0) )) * k1
                        - Sqr((2*i+1.0)/(2*i-3.0)) * Sqr(k/(k-1.0)) * k0;
                   k0 = k1; k1 =k2;
                } break;
       }
 return k2;
}

//*******class hGradient : public virtual Functions*************

/**************************************************************************
  **                    Procedure hGradientsCalculation:                   **
  **        Computation of the derivatives of h with regard to x           **
  *************************************************************************
*/

void hGradient::hGradientCalculation()
{
 int f,m;
 double h1,h2,h3,h4,h5,h6,T1,T2,T3,T4;

// Used are the recursion relations T(n) = 2 x T(n-1) - T(n-2)
// and T'(n)/n = 2 T(n-1) + T'(n-2)/(n-2).

 for(f=0;f<nk;f++){
    T1=x[f]; T2=1; T3=x[f]; T4=2*x[f]*x[f]-1;
    h1=4*T3; h2=2*T2-1; h3=4*h2; h4=0; h5=0; h6=h1;
    hx[f]=b[0]+b[1]*T4; dhdx[f]=b[1]*h1; d2hdx2[f]=b[1]*h3; d3hdx3[f]=0;

    for(m=2;m<=Ncoef_b;m++){
       T1 = T3; T2 = T4; T3 = 2 * x[f] * T2 - T1; T4 = 2 * x[f] * T3 - T2;
       hx[f] += b[m] * T4;
       h1 = m * (4 * T3 + h1/(m - 1));
       dhdx[f] += b[m] * h1;
       h2 = (2 * m - 1) * (2 * T2 + h2/(2 * m - 3));
       h3 = m * (4 * h2 + h3/(m - 1));
       d2hdx2[f] += b[m] * h3;
       h4 = (2 * m - 1) * (2 * h6 + h4/(2 * m - 3));
       h5 = m * (4 * h4 + h5/(m - 1));
       d3hdx3[f] += b[m] * h5;
       h6 = h1;
    }
 }
}

//------------------- End of hGradientsCalculation ----------------------

//*********class Functions : public virtual Data*********

/*************************************************************************
  ** Function h(Xi)                                                     **
  ************************************************************************
*/

double Functions::funct_h(double xi)
{
 int m;
 register double help,T1,T2,T3,T4;
// Here the height h of the liquid is calculated.

 help = b[0]; T1 = T2 = 0; T3 = xi; T4 = 1;
 for(m=1;m<=Ncoef_b;m++){
    T1 = T3; T2 = T4; T3 = 2 * xi * T2 - T1; T4 = 2 * xi * T3 - T2;
    help += b[m] * T4;
 }
 return help;
}
//---------------------- End of funct_h----------------------------------

/*************************************************************************
  ** Function dhdx(Xi)                                                  **
  ************************************************************************
*/

double Functions::funct_dhdx(double xi)
{
 int m;
 register double help,T1,T2,T3,T4,h1;
// Here the derivative of h with respect to x is calculated.
```

```
 T1 = xi; T2 = 1; T3 = xi; T4 = 2*xi*xi-1;
 h1 = 4 * T3; help = b[1] * h1;
 for(m=2;m<=Ncoef_b;m++){
    T1 = T3; T2 = T4; T3 = 2 * xi * T2 - T1; T4 = 2 * xi * T3 - T2;
    h1 = m * (4 * T3 + h1/(m - 1));
    help += b[m] * h1;
 }
 return help;
}
//---------------------- End of funct_dhdx-------------------------------

/*************************************************************************
  ** Function d2hdx2(Xi)                                                **
  ************************************************************************
*/

double Functions::funct_d2hdx2(double xi)
{
 int m;
 register double help,h2,h3,T1,T2,T3,T4;
// Here the second derivative of h with respect to x is calculated.

 T1=xi; T2=1; T3=xi; T4=2*xi*xi-1;
 h2=2*T2-1; h3=4*h2;
 help=b[1]*h3;

 for(m=2;m<=Ncoef_b;m++){
    T1 = T3; T2 = T4; T3 = 2 * xi * T2 - T1; T4 = 2 * xi * T3 - T2;
    h2 = (2 * m - 1) * (2 * T2 + h2/(2 * m - 3));
    h3 = m * (4 * h2 + h3/(m - 1));
    help += b[m] * h3;
 }
 return help;
}
//---------------------- End of funct_d2hdx2-----------------------------
-

/*************************************************************************
  ** Function d3hdx3(Xi)                                                **
  ************************************************************************
*/

double Functions::funct_d3hdx3(double xi)
{
 int m;
 register double help,h1,h4,h5,h6,T1,T2,T3,T4;
// Here the third derivative of h with respect to x is calculated.

 T1=xi; T2=1; T3=xi; T4=2*xi*xi-1;
 h1=4*T3; h4=0; h5=0; h6=h1;
 help=0;

 for(m=2;m<=Ncoef_b;m++){
    T1 = T3; T2 = T4; T3 = 2 * xi * T2 - T1; T4 = 2 * xi * T3 - T2;
    h1 = m * (4 * T3 + h1/(m - 1));
    h4 = (2 * m - 1) * (2 * h6 + h4/(2 * m - 3));
    h5 = m * (4 * h4 + h5/(m - 1));
    help += b[m] * h5;
    h6 = h1;
 }
 return help;
}
//----------------------- End of funct_d3hdx3-------------------------------
-

void New_DB::NewH()
{
 int i,j,k,mx,z2,q,m;
 double del,xv,nor,hc1,ef,ha,dhadx,dhadt;
 double T1,T2,T3,T4,h0,h1;
 mx = Ncoef_b;
 if(Ncoef_b<nic) mx = nic;
 if(mx<nkc) mx = nkc;
 vector bnn(Ncoef_b+1);

 z2  = 512;
 q   = 5;
 del = 1.0/z2;
 xv  = -del;
 for(j=0;j<=z2;j++){
    q=6-q;
    if(j==z2) q=1;
    nor = q * del / 3;
    xv += del;


    T1=xv; T2=1; T3=xv; T4=2*xv*xv-1;
    h1=4*T3; ha=b[0]+b[1]*T4; dhadx=b[1]*h1;

    for(i=2;i<=Ncoef_b;i++){
       T1 = T3; T2 = T4; T3 = 2 * xv * T2 - T1; T4 = 2 * xv * T3 - T2;
       ha += b[i] * T4;
       h1 = i * (4 * T3 + h1/(i - 1));
       dhadx += b[i] * h1;
    }

    for(dhadt=0.0,k=0;k<nkc;k++){
       h0 = fc0(k,xv) * dhadx;
       h1 = -ha*fc1(k,xv) - h0;
       for(i=0;i<nic;i++) dhadt += d[i*nkc+k] * (h0*gc0(i,1) + h1*gg(i,1) -
 h0);
    }

    ef = ha - 1.0 + dhadt * timestep;

    T1 = T2 = 0; T3 = xv; T4 = 1;
    for(m=1;m<=Ncoef_b;m++){
       T1 = T3; T2 = T4; T3 = 2 * xv * T2 - T1; T4 = 2 * xv * T3 - T2;
```

```
        bnn[i] += nor * ef * T4;
     }
     if(j==0) q=2;
 }

 for(i=1;i<=Ncoef_b;i++) b[i] = bnn[i];
}
```

APPENDIX

```
/************************************************************************
***
 ***   Procedure ContourInt computes the integrals, the matrix elements
***
 ***                   and the coefficients of e[i,k]
***
 ***
***
 ***            (P1-P2) + (P2-P3) + (P3-P4) + (P4-P0) + (P0-P1) = 0
***
 ***********************************************************************
***

Before solving the time evolution, the Navier-Stokes equations have to be
solved. Consequently a great number of integrals have to be solved.
In fact, for every collocation point there has to be integrated along a
contour from the collocation point to the wall, along a line of constant
c, then to the surface of the liquid and along a vertical line back to the
collocation point.
*/

//**class TimeStep : public Contour_Int_New, public New_DB, public hGradien
t**

void TimeStep::ContourInt()
{
 /*
   Here the matrix ah and the vector bvec are composed for the computation
   of the time derivatives of the d values, i.e. the e values, for the
   collocation points situated between 0<x<1 and 0<y<1.
  */

 int ff,q,i1,g,i,j,k,ik,l,il;
 double Ix_dhdx_d2vdx2,Iy_d2v_dy2,Iy_vdvdy,Iy_d2v_dx2_xis1,coprex,h4,h6,h7,
        Ix_dhdx_d2vdy2,Ix_ududx,Ix_vdudy,Ix_dhdx_udvdx,Ix_d2u_dx2,h1,h2,d2,
        Ix_dhdx_vdvdy,Ix_d2u_dy2,d2hdx2_xis1,v2_xis1,p0_p1,p4_p0,p1_p2,pik,
        p2_p3,p4_p1,p3_p4[nimax],alfap_1,alfap_2,alfap_3,xl,sqraver,cpowi,
        ii3[nkcmax],ii5[nkcmax],ii7[nkcmax],ii4[nkcmax],ii8[nkcmax],p1,p2,h
3,
        h11,z2,nor,diff,sqrpi,co,si,cosk[ncoefmax+1],sink[ncoefmax+1],d3,
        sumu,sum1dudx,sum2dudx,sum1dvdx,sum2dvdx,sum3dvdx,sum4dvdx,sum1v,hh
,
        sum2v,sum1d2vdx2,sum2d2vdx2,sum3d2vdx2,sum4d2vdx2,sum5d2vdx2,dd,
        sum6d2vdx2,sum7d2vdx2,c2,Iy_udvdx[nkmax],Iy_dvdt[nkmax],diffc,qc,
        Iy_d2vdx2,delc,cv,cvc,Ix_dudt,Ix_dhdx_dvdt,P_v,P_dudx,P_dvdx,x1,x2,
x3,
        sums1[nicmax],sums2[nicmax],sums3[nicmax],sums4[nicmax],x4,x5,x6,
        x7,del,sum1[nkcmax],sum2[nkcmax],sum3[nkcmax],sum4[nkcmax],sum1dhdt
,
        sum5[nkcmax],sum6[nkcmax],cpow,ssum1,ssum2,ssum3,ssum4,x8,ii2[nkcma
x],
        sum2dhdt,sum3dhdt;
 register double rd,ac,as,norm,mx;
 cout << xy(10,9) << cleol;
 cout << xy(39,14) << "   " << xy(39,14) << ntimesteps-step+1;

 sqraver = aver * aver;
 sqrpi = pi * pi;

 //--- calculation of p0-p4 ---
 d2hdx2_xis1= funct_d2hdx2(1);

 //***   p0-p4  ***

 p4_p0 = -2.0 * sqraver * funct_dvdy(1,1,1,0) + sqraver * sgr_xis1
        * d2hdx2_xis1;

//   Due to the no slip condition at the wall h_xis1 = 1

 //-- calculation of p4-p3 ---

 for(g=0;g<ni;p3_p4[g]=Sqr(sqraver)*Iy_d2v_dx2_xis1,g++){
    h1=c[g] * c[g];
    for(Iy_d2v_dx2_xis1=0,i=0;i<nic;i++){
       h1 *= c[g];
       for(k=0;k<nkc;k++){
          ik          = i*nkc+k;
          v2_xis1          = (3*(i+1)*d2hdx2_xis1 + sqrpi*(k+1)*(k+1))
                            *pi*(k+1)*OddNeg(k+1)/(i+2);
          Iy_d2v_dx2_xis1+= d[ik] * (1-h1) * v2_xis1/(i+3);
       }
    }
 }
 // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 -

 for(i=0;i<nk;i++) Iy_udvdx[i]  = Iy_dvdt[i] = 0.0;
 for(g=0;g<ni;g++){

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    cout << xy(47,14) << "  " << xy(47,14) << ni-g;

 // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    if(g==0)
      diffc = 1-c[g];
    else
      diffc = c[g-1] - c[g];
       h1 = part * diffc / (1 - c[ni-1]);
       c2 = floor(h1); c2 += (1 - OddNeg(c2)) / 2;
    delc = diffc / c2;
    cv = c[g] + diffc;

    for(k=0;k<nkc;k++){
       ii2[k]=ii3[k]=ii4[k]=ii5[k]=ii7[k]=ii8[k]=0;
       sum1[k] = sum2[k] = sum3[k] = sum4[k] = sum5[k] = sum6[k] = 0;
       cpowi = 1.0;
```

```
        for(i=0;i<nic;i++){
           ik = i * nkc + k;
           cpowi *= c[g];
           sum1[k] += d[ik] * cpowi;
           sum2[k] += d[ik] * (i+1) * cpowi;
           sum3[k] += d[ik] * (i+1) / (i+2) * cpowi;
           sum4[k] += d[ik] / (i+2) * cpowi;
           sum5[k] += d[ik] * (i+1) * (i+3) * cpowi;
           sum6[k] += d[ik] / (i+2);
        }
     }

     Ix_ududx        = 0;
     Ix_vdudy        = 0;
     Ix_dhdx_vdvdy   = 0;
     Ix_dhdx_udvdx   = 0;
     Ix_dhdx_d2vdx2  = 0;
     Ix_dudt         = 0;
     Ix_dhdx_dvdt    = 0;
     for(ff=0;ff<nk;ff++){

        mx = 0;
        Iy_d2vdx2 = 0.0;
        q=5;
        if(ff==0)
          diff = 1-x[ff];
        else
            diff = x[ff-1] - x[ff];
        h1 = part * diff / (1 - x[nk-1]);
        z2 = floor(h1); z2 += (1 - OddNeg(z2)) / 2;
        del=pi*diff/z2;
        p2=sin(del); p1=cos(del);
        ac=cos(pi*(x[ff]+diff)); as=sin(pi*(x[ff]+diff));
        norm = diff/(3*z2);
        for(j=0;j<=z2;j++){
           q=6-q;
           co = 1.0; si = 0.0;
           hh = 1; dd = d2 = d3 = 0;
           i1 = 0;
           for(i=0;i<Ncoef_b;i++){
              cosk[i] = co*ac-si*as;
              si = si*ac+co*as;
              sink[i] = si;
              co = cosk[i];
              h1 = b[i+1] * co;
              h2 = b[i+1] * si * (i+1);
              hh += h1;
              dd += h2;
              i1 += i + i + 1;
              d2 += h1 * i1;
              d3 += h2 * i1;
           }
           dd *= -pi; d2 *= -pi * pi; d3 *= pi * pi * pi;
           for(i=Ncoef_b;i<nkc;i++){
```

```
      cosk[i] = co*ac-si*as;
      si = si*ac+co*as;
      sink[i] = si;
      co = cosk[i];
   }
   if(j==z2) q=1;
   nor=q*del/(pi*3);
   x1=dd/hh;
   x2=dd*dd;
   x3=nor*hh*dd;
   x4=nor*x1*x1;
   x5=nor*x2;
   x6=hh*hh/nor;
   x7=nor*x1;
   x8=nor*d2/hh;
   for(k=0;k<nkc;k++){
      ii2[k]+=sink[k]*x8;
      ii3[k]+=sink[k]*x4;
      ii5[k]+=sink[k]/x6;
      ii7[k]+=sink[k]*x5;
      ii4[k]+=cosk[k]*x7;
      ii8[k]+=cosk[k]*x3;
   }

   sumu = sum1dudx = sum2dudx = sum1dvdx = 0;
   sum2dvdx = sum1v = sum2v = sum1d2vdx2 = 0;
   sum2d2vdx2 = sum3d2vdx2 = sum4d2vdx2 =  0;
   sum1dhdt = sum2dhdt = sum3dhdt = 0;
   for(k=0;k<nkc;k++){
      sumu        += sum1[k] * sink[k];
      sum1dudx    += sum1[k] * cosk[k] * (k+1);
      sum2dudx    += sum2[k] * sink[k];
      sum1v       += sum4[k] * cosk[k] * (k+1);
      sum2v       += sum3[k] * sink[k];
      sum1dhdt    += sum6[k] * sink[k];
      sum2dhdt    += sum6[k] * cosk[k] * (k+1);
      sum3dhdt    += sum6[k] * sink[k] * (k+1) * (k+1);
      sum1dvdx    += sum4[k] * sink[k] * (k+1) * (k+1);
      sum2dvdx    += sum3[k] * cosk[k] * (k+1);
      sum1d2vdx2  += sum4[k] * cosk[k] * (k+1) * (k+1) * (k+1);
      sum2d2vdx2  += sum3[k] * sink[k] * (k+1) * (k+1);
      sum3d2vdx2  += sum2[k] * cosk[k] * (k+1);
      sum4d2vdx2  += sum5[k] * sink[k];
   }
     P_dvdx           = q * c[g] * (sqrpi * hh * sum1dvdx
                    + 2 * dd * pi * sum2dvdx + d2 * sum2v -
                    sum2dudx * dd * x1);
   P_dudx           = q * (pi * sum1dudx - sum2dudx * x1);
   P_v              = q * c[g] * (sum2v * dd - pi * hh * sum1v);
   Ix_ududx        += norm * sumu * P_dudx;
   Ix_vdudy        += norm * P_v * sum2dudx / ( c[g] * hh );
   Ix_dhdx_vdvdy   += -norm * dd * P_v * P_dudx / q;
   Ix_dhdx_udvdx   += norm * dd * sumu * P_dvdx;
   Ix_dhdx_d2vdx2  += norm * q * c[g] * dd * (hh * sqrpi * pi *
                                  sum1d2vdx2 -3 * dd * sqrpi * sum2d2vdx2 + 3 * d
2
                                  * pi * sum2dvdx  - 3 * dd * x1 * pi * sum3d2vdx
2 +
                                  d3 * sum2v - 3 * d2 * x1 * sum2dudx + x1 * x1
                                  * dd * sum4d2vdx2);
          Ix_dudt           += norm * q * (sum1dhdt * dd + sum2dhdt * pi * hh)
 *
                                  sum2dudx / hh;
          Ix_dhdx_dvdt      += norm * q * c[g] * dd * ((-sum1dhdt * dd - sum2d
hdt
                                  * pi * hh) * (sum2dvdx * pi -sum2dudx * x1) +
                                  (-2.0 * sum2dhdt * pi * dd + sum3dhdt * pi * pi
 *
                                  hh - sum1dhdt * d2) * sum2v);

          h1=ac*p1+as*p2;
          as=as*p1-ac*p2;
          ac=h1;
          if(j==0) q=2;
       }
       Ix_d2u_dx2      = 0;
       Ix_d2u_dy2      = 0;
       Ix_dhdx_d2vdy2 = 0;
       for(k=0;k<nkc;k++){
          pik      = pi * (k+1);
          cpowi    = 1.0;
          for(i=0;i<nic;i++){
             ik = i * nkc + k;
             cpowi   *= c[g];
             h6=d[ik]*cpowi;
             Ix_d2u_dx2 += h6 * (-pik * (OddNeg(k) + cosk[k] + 2.0 * (i+1)
*
                                 ii4[k]) - (i+1) * ii2[k] + (i+1) * (i+2) * ii3[k
]);
             Ix_d2u_dy2 += h6 * (i+1) * i * ii5[k] / (c[g] * c[g]);
             Ix_dhdx_d2vdy2 += h6 * (i+1) * ((i+1) * ii3[k] - pik*ii4[k])/
                                 c[g];
          }
           }
       sum1d2vdx2 = sum2d2vdx2 = sum3d2vdx2 = sum4d2vdx2 = sum5d2vdx2 =
       sum6d2vdx2 = sum7d2vdx2 = 0;

       for(i=0;i<nic;i++){
          ik = i * nkc - 1;
          for(ssum1=ssum2=ssum3=ssum4=0,k=0;k<nkc;k++){
                ++ik;
                ssum1 += d[ik] * sink[k];
                ssum2 += d[ik] * cosk[k] * (k+1);
                ssum3 += d[ik] * sink[k] * (k+1) * (k+1);
                ssum4 += d[ik] * cosk[k] * (k+1) * (k+1) * (k+1);
          }
          sums1[i] = ssum1;
          sums2[i] = ssum2;
```

```
            sums3[i] = ssum3;
            sums4[i] = ssum4;
         }
         cvc = cv;
         qc=5;
         for(j=0;j<=c2;j++){
            qc=6-qc;
            if(j==c2) qc=1;
            nor=qc*delc/3;
            cpowi=1.0;
            sumu = sum1dvdx = sum2dvdx = sum3dvdx = sum4dvdx = sum1v = sum2v
            = sum1dhdt = sum2dhdt = sum3dhdt = 0;
            for(i=0;i<nic;i++){
               cpowi *= cvc;
               sumu     += cpowi * sums1[i];
               sum1dvdx += cpowi * sums3[i] / (i+2);
               sum2dvdx += cpowi * sums2[i] * (i+1) / (i+2);
               sum3dvdx += cpowi * sums1[i] * (i+1) / (i+2);
               sum4dvdx += cpowi * sums1[i] * (i+1);
               sum1dhdt += sums1[i] / (i+2);
               sum2dhdt += sums2[i] / (i+2);
               sum3dhdt += sums3[i] / (i+2);
               sum1v    += cpowi * sums2[i] / (i+2);
               sum2v    += cpowi * sums2[i];
               if(j==c2){
                 cpow = (1 - cpowi * cvc * cvc)/(i+3);
                 sum1d2vdx2 += cpow * sums4[i] / (i+2);
                 sum2d2vdx2 += cpow * sums3[i] * (i+1) / (i+2);
                 sum3d2vdx2 += cpow * sums2[i] * (i+1) / (i+2);
                 sum4d2vdx2 += cpow * sums2[i] * (i+1);
                 sum5d2vdx2 += cpow * sums1[i] * (i+1) / (i+2);
                 sum6d2vdx2 += cpow * sums1[i] * (i+1);
                 sum7d2vdx2 += cpow * sums1[i] * (i+1) * (i+3);
               }
                }
            P_dvdx          = cvc * (sqrpi * hh * sum1dvdx + 2 *
                              pi * dd * sum2dvdx + d2 * sum3dvdx - dd * x1 *
                              sum4dvdx);
            Iy_udvdx[ff] += nor * hh * sumu * P_dvdx;
            P_v             = -sum1dhdt * dd - sum2dhdt * pi * hh;
            Iy_dvdt[ff]  += nor * cvc * hh * (P_v * (pi * sum2dvdx
                              - x1 * sum4dvdx) + (-2.0 * sum2dhdt * pi * dd +
                              sum3dhdt * pi * pi * hh - sum1dhdt * d2) * sum3dv
dx);
            if(j==c2){
              Iy_d2vdx2 = hh * (sum1d2vdx2 * sqrpi * pi * hh - 3 *
                          sum2d2vdx2 * sqrpi * dd + 3 * sum3d2vdx2 * pi * d2
-
                          3 * sum4d2vdx2 * pi * dd * x1 + sum5d2vdx2 * d3 - 3
 *
                          sum6d2vdx2 * d2 * x1 + sum7d2vdx2 * dd * x1 * x1);
            }
            cvc -= delc;
            if(j==0) qc=2;
```

```
       }

       // ***  p1-p0  ***

       p0_p1 = prss[ff];

       p4_p1 = p4_p0 + p0_p1;

       p2_p3 = Ix_d2u_dy2 + sqraver * Ix_d2u_dx2 - re * aver * (Ix_dudt +
               Ix_ududx + Ix_vdudy) + c[g] * sqraver * ( Ix_dhdx_d2vdy2 +
               sqraver * Ix_dhdx_d2vdx2 - re * aver * (Ix_dhdx_dvdt +
               Ix_dhdx_udvdx + Ix_dhdx_vdvdy));

      //--- calculation of p2-p1 ---
      Iy_d2v_dy2 = funct_dvdy(x[ff],1,hh,dd)
                 - funct_dvdy(x[ff],c[g],hh,dd);

      coprex= Sqr(funct_v(x[ff],1,hh,dd));
      Iy_vdvdy=funct_v(x[ff],c[g],hh,dd);
      Iy_vdvdy=0.5*(coprex-Sqr(Iy_vdvdy));


      //*** (p2-p1)***

      coprex = sqraver * Iy_d2v_dy2 + Sqr(sqraver) * Iy_d2vdx2;
      p1_p2  = -coprex + re*sqraver*aver * (Iy_dvdt[ff] +
               Iy_udvdx[ff] + Iy_vdvdy);

      rd = p1_p2 + p2_p3 + p3_p4[g] + p4_p1;

         //calculate the alfa elements
      for(k=0;k<nkc;k++){
         cpowi = 1.0;
         for(i=0;i<nic;i++){

            cpowi *= c[g];
         // Ix_dudt
            alfap_1 = re * aver * cpowi * (OddNeg(k) + cosk[k]) / (pi*(k+1)
);
         // Ix_dhdx_dvdt
            coprex  = (i+1) * ii7[k] - pi * (k+1) * ii8[k];
            alfap_2= re*sqraver*aver*cpowi*c[g]*c[g]
                     * coprex/(i+2.0);
         // Iy_dvdt
            coprex= ((i+1) * sink[k] * dd -pi * (k+1) * hh * cosk[k]) *
                    (-re*sqraver*aver*hh);
            alfap_3= coprex*(1.0 - cpowi * c[g] * c[g])/((i+2)*(i+3.0));
         // alfa-elements
            if(k<nkc-1) helpa[i * (nkc-1) + k] = alfap_1 + alfap_2 + alfap_
3;
            else{
               h1 = alfap_1 + alfap_2 + alfap_3;
               for(l=0;l<nkc-1;l++){
                  il = i * (nkc-1) + l;
```

```
                    helpa[il] += h1 * (l+1) * OddNeg(nkc+l) / nkc;
                    if(fabs(helpa[il])>mx) mx = fabs(helpa[il]);
                }
            }
        }
      }

      mx *= mx;
      for(k=0;k<(nkc-1)*nic;k++){
         bvec[k] += rd * helpa[k] / mx;
         for(i=0;i<nic*(nkc-1);i++)
            ah[k][i] += helpa[k] * helpa[i] / mx;
      }
    }     // end of ff loop
 }        // end of g loop
}

//-------------------------- End of ContourInt ---------------------------
-
```

```
/***************************************************************************
***
 ***   Procedure ContourInt computes the integrals, the matrix elements
***
 ***                 and the coefficients of e[i,k]
***
 ***
***
 ***            (P1-P2) + (P2-P3) + (P3-P4) + (P4-P0) + (P0-P1) = 0
***
 ***************************************************************************
***

Before solving the time evolution, the Navier-Stokes equations have to be
solved. Consequently a great number of integrals have to be solved.
In fact, for every collocation point there has to be integrated along a
contour from one collocation point to an adjacent collocation point,
along a line of constant c-value, then to the surface of the liquid and
along a vertical line back to the first mentioned collocation point.
*/

//****class TimeStep : public Contour_Int_New, public New_DB***

void TimeStep::ContourInt()
{
 /*
   Here the matrix ah and the vector bvec are composed for the computation
   of the time derivatives of the d values, i.e. the e values, for the
   collocation points situated between 0<x<1 and 0<y<1.
  */

 int ff,q,i1,g,i,j,k,ik,l,il;
 double Ix_dhdx_d2vdx2,Iy_d2v_dy2,Iy_vdvdy,Iy_d2v_dx2_xis1,coprex,h4,h6,h7,
        Ix_dhdx_d2vdy2,Ix_ududx,Ix_vdudy,Ix_dhdx_udvdx,Ix_d2u_dx2,h1,h2,d2,
        Ix_dhdx_vdvdy,Ix_d2u_dy2,d2hdx2_xis1,v2_xis1,p0_p1,p4_p0,p1_p2,pik,
        p2_p3,p4_p1,p3_p4[nimax],alfap_1,alfap_2,alfap_3,xl,sqraver,cpowi,
        ii3[nkcmax],ii5[nkcmax],ii7[nkcmax],ii4[nkcmax],ii8[nkcmax],p1,p2,h
3,
        h11,z2,nor,diff,sqrpi,co,si,cosk[ncoefmax+1],sink[ncoefmax+1],d3,
        sumu,sum1dudx,sum2dudx,sum1dvdx,sum2dvdx,sum3dvdx,sum4dvdx,sum1v,hh
,
        sum2v,sum1d2vdx2,sum2d2vdx2,sum3d2vdx2,sum4d2vdx2,sum5d2vdx2,dd,p34
,
        sum6d2vdx2,sum7d2vdx2,c2,Iy_udvdx[nkmax],Iy_dvdt[nkmax],diffc,qc,
        Iy_d2vdx2,delc,cv,cvc,Ix_dudt,Ix_dhdx_dvdt,P_v,P_dudx,P_dvdx,x1,x2,
x3,
        sums1[nicmax],sums2[nicmax],sums3[nicmax],sums4[nicmax],x4,x5,x6,
        x7,del,sum1[nkcmax],sum2[nkcmax],sum3[nkcmax],sum4[nkcmax],
        sum5[nkcmax],cpow,ssum1,ssum2,ssum3,ssum4,x8,,ii2[nkcmax],ii6[nkcma
x],
        sum6[nkcmax],sum1dhdt,sum2dhdt,sum3dhdt;
 register double rd,ac,as,norm,mx;

 cout << xy(10,9) << cleol;
 cout << xy(39,14) << "    " << xy(39,14) << ntimesteps-step+1;

 sqraver = aver * aver;
 sqrpi = pi * pi;

 //--- calculation of p0-p4 ---
 d2hdx2_xis1= funct_d2hdx2(1);

 //***   p0-p4   ***

 p4_p0 = -2.0 * sqraver * funct_dvdy(1,1,1,0) + sqraver * sgr_xis1
         * d2hdx2_xis1;

 //-- calculation of p4-p3 along the wall---

 for(g=0;g<ni;p3_p4[g]=Sqr(sqraver)*Iy_d2v_dx2_xis1,g++){
    h1=c[g] * c[g];
    for(Iy_d2v_dx2_xis1=0,i=0;i<nic;i++){
       h1 *= c[g];
       for(k=0;k<nkc;k++){
          ik          = i*nkc+k;
          v2_xis1          = (3*(i+1)*d2hdx2_xis1 + sqrpi*(k+1)*(k+1))
                             *pi*(k+1)*OddNeg(k+1)/(i+2);
          Iy_d2v_dx2_xis1+= d[ik] * (1-h1) * v2_xis1/(i+3);
       }
    }
 }
 // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 -

 for(i=0;i<nk;i++) Iy_udvdx[i]  = Iy_dvdt[i] = 0.0;
 for(g=0;g<ni;g++){

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    cout << xy(47,14) << "  " << xy(47,14) << ni-g;

 // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    if(g==0)
      diffc = 1-c[g];
    else
      diffc = c[g-1] - c[g];
       h1 = part * diffc / (1 - c[ni-1]);
       c2 = floor(h1); c2 += (1 - OddNeg(c2)) / 2;
       delc = diffc / c2;
       cv = c[g] + diffc;

    for(k=0;k<nkc;k++){
       sum1[k] = sum2[k] = sum3[k] = sum4[k] = sum5[k] = sum6[k] = 0;
       cpowi = 1.0;
       for(i=0;i<nic;i++){
          ik = i * nkc + k;
          cpowi *= c[g];
          sum1[k] += d[ik] * cpowi;
          sum2[k] += d[ik] * (i+1) * cpowi;
          sum3[k] += d[ik] * (i+1) / (i+2) * cpowi;
          sum4[k] += d[ik] / (i+2) * cpowi;
          sum5[k] += d[ik] * (i+1) * (i+3) * cpowi;
          sum6[k] += d[ik] / (i+2);
       }
    }

    for(ff=0;ff<nk;ff++){
       for(k=0;k<nkc;k++) ii2[k]=ii3[k]=ii4[k]=ii5[k]=ii6[k]=ii7[k]=ii8[k]=
0;
       Ix_ududx       = 0;
       Ix_vdudy       = 0;
       Ix_dhdx_vdvdy  = 0;
       Ix_dhdx_udvdx  = 0;
       Ix_dhdx_d2vdx2 = 0;
       Ix_dudt        = 0;
       Ix_dhdx_dvdt   = 0;
       mx = 0;
       Iy_d2vdx2 = 0.0;
       q=5;
       if(ff==0)
         diff = 1-x[ff];
       else
           diff = x[ff-1] - x[ff];
       h1 = part * diff / (1 - x[nk-1]);
       z2 = floor(h1); z2 += (1 - OddNeg(z2)) / 2;
       del=pi*diff/z2;
       p2=sin(del); p1=cos(del);
       ac=cos(pi*(x[ff]+diff)); as=sin(pi*(x[ff]+diff));
       norm = diff/(3*z2);

       for(j=0;j<=z2;j++){
          q=6-q;
          co = 1.0; si = 0.0;
          hh = 1; dd = d2 = d3 = 0;
          i1 = 0;
          for(i=0;i<Ncoef_b;i++){
             cosk[i] = co*ac-si*as;
             si = si*ac+co*as;
             sink[i] = si;
             co = cosk[i];
             h1 = b[i+1] * co;
             h2 = b[i+1] * si * (i+1);
             hh += h1;
             dd += h2;
             i1 += i + i + 1;
             d2 += h1 * i1;
             d3 += h2 * i1;
          }
          dd *= -pi; d2 *= -pi * pi; d3 *= pi * pi * pi;
          for(i=Ncoef_b;i<nkc;i++){
             cosk[i] = co*ac-si*as;
             si = si*ac+co*as;
```

```
            sink[i] = si;
            co = cosk[i];
         }

         if(j==z2) q=1;
         nor=q*del/(pi*3);
         x1=dd/hh;
         x2=dd*dd;
         x3=nor*hh*dd;
         x4=nor*x1*x1;
         x5=nor*x2;
         x6=hh*hh/nor;
         x7=nor*x1;
         x8=nor*d2/hh;
         for(k=0;k<nkc;k++){
            ii6[k]+=sink[k]*nor;
            ii2[k]+=sink[k]*x8;
            ii3[k]+=sink[k]*x4;
            ii5[k]+=sink[k]/x6;
            ii7[k]+=sink[k]*x5;
            ii4[k]+=cosk[k]*x7;
            ii8[k]+=cosk[k]*x3;
         }

         sumu = sum1dudx = sum2dudx = sum1dvdx = 0;
         sum2dvdx = sum1v = sum2v = sum1d2vdx2 = 0;
         sum2d2vdx2 = sum3d2vdx2 = sum4d2vdx2 =  0;
         sum1dhdt = sum2dhdt = sum3dhdt = 0;
         for(k=0;k<nkc;k++){
            sumu        += sum1[k] * sink[k];
            sum1dudx    += sum1[k] * cosk[k] * (k+1);
            sum2dudx    += sum2[k] * sink[k];
            sum1v       += sum4[k] * cosk[k] * (k+1);
            sum2v       += sum3[k] * sink[k];
            sum1dhdt    += sum6[k] * sink[k];
            sum2dhdt    += sum6[k] * cosk[k] * (k+1);
            sum3dhdt    += sum6[k] * sink[k] * (k+1) * (k+1);
            sum1dvdx    += sum4[k] * sink[k] * (k+1) * (k+1);
            sum2dvdx    += sum3[k] * cosk[k] * (k+1);
            sum1d2vdx2 += sum4[k] * cosk[k] * (k+1) * (k+1) * (k+1);
            sum2d2vdx2 += sum3[k] * sink[k] * (k+1) * (k+1);
            sum3d2vdx2 += sum2[k] * cosk[k] * (k+1);
            sum4d2vdx2 += sum5[k] * sink[k];
         }
             P_dvdx          = q * c[g] * (sqrpi * hh * sum1dvdx
                           + 2 * dd * pi * sum2dvdx + d2 * sum2v -
                           sum2dudx * dd * x1);
         P_dudx           = q * (pi * sum1dudx - sum2dudx * x1);
         P_v              = q * c[g] * (sum2v * dd - pi * hh * sum1v);
         Ix_udux         += norm * sumu * P_dudx;
         Ix_vdudy        += norm * P_v * sum2dudx / ( c[g] * hh );
         Ix_dhdx_vdvdy   += -norm * dd * P_v * P_dudx / q;
         Ix_dhdx_udvdx   += norm * dd * sumu * P_dvdx;
         Ix_dhdx_d2vdx2 += norm * q * c[g] * dd * (hh * sqrpi * pi *
                                 sum1d2vdx2 -3 * dd * sqrpi * sum2d2vdx2 + 3 * d
2
                                 * pi * sum2dvdx  - 3 * dd * x1 * pi * sum3d2vdx
2 +
                                 d3 * sum2v - 3 * d2 * x1 * sum2dudx + x1 * x1
                                 * dd * sum4d2vdx2);
          Ix_dudt         += norm * q * (sum1dhdt * dd + sum2dhdt * pi * hh)
 *
                                 sum2dudx / hh;
          Ix_dhdx_dvdt    += norm * q * c[g] * dd * ((-sum1dhdt * dd - sum2d
hdt
                                 * pi * hh) * (sum2dvdx * pi -sum2dudx * x1) +
                                 (-2.0 * sum2dhdt * pi * dd + sum3dhdt * pi * pi
 *
                                 hh - sum1dhdt * d2) * sum2v);
          h1=ac*p1+as*p2;
          as=as*p1-ac*p2;
          ac=h1;
          if(j==0) q=2;
       }

       Ix_d2u_dx2     = 0;
       Ix_d2u_dy2     = 0;
       Ix_dhdx_d2vdy2 = 0;
       for(k=0;k<nkc;k++){
          pik     = pi * (k+1);
          cpowi   = 1.0;
          for(i=0;i<nic;i++){
             ik = i * nkc + k;
             cpowi  *= c[g];
             h6=d[ik]*cpowi;
             Ix_d2u_dx2 += h6 * (-pik * pik * ii6[k] -2.0 * pik * (i+1) *
                            ii4[k] - (i+1) * ii2[k] + (i+1) * (i+2) * ii3[k]
);
             Ix_d2u_dy2 += h6 * (i+1) * i * ii5[k] / (c[g] * c[g]);
             Ix_dhdx_d2vdy2 += h6 * (i+1) * ((i+1) * ii3[k] - pik*ii4[k])/
                            c[g];
          }
           }

       sum1d2vdx2 = sum2d2vdx2 = sum3d2vdx2 = sum4d2vdx2 = sum5d2vdx2 =
       sum6d2vdx2 = sum7d2vdx2 = 0;

       for(i=0;i<nic;i++){
          ik = i * nkc - 1;
          for(ssum1=ssum2=ssum3=ssum4=0,k=0;k<nkc;k++){
                ++ik;
                ssum1 += d[ik] * sink[k];
                ssum2 += d[ik] * cosk[k] * (k+1);
                ssum3 += d[ik] * sink[k] * (k+1) * (k+1);
                ssum4 += d[ik] * cosk[k] * (k+1) * (k+1) * (k+1);
          }
          sums1[i] = ssum1;
          sums2[i] = ssum2;
```

```
            sums3[i] = ssum3;
            sums4[i] = ssum4;
        }

        cvc = cv;
        qc=5;
        for(j=0;j<=c2;j++){
            qc=6-qc;
            if(j==c2) qc=1;
            nor=qc*delc/3;
            cpowi=1.0;
            sumu = sum1dvdx = sum2dvdx = sum3dvdx = sum4dvdx = sum1v = sum2v
            = sum1dhdt = sum2dhdt = sum3dhdt = 0;
            for(i=0;i<nic;i++){
                cpowi *= cvc;
                sumu     += cpowi * sums1[i];
                sum1dvdx += cpowi * sums3[i] / (i+2);
                sum2dvdx += cpowi * sums2[i] * (i+1) / (i+2);
                sum3dvdx += cpowi * sums1[i] * (i+1) / (i+2);
                sum4dvdx += cpowi * sums1[i] * (i+1);
                sum1dhdt += sums1[i] / (i+2);
                sum2dhdt += sums2[i] / (i+2);
                sum3dhdt += sums3[i] / (i+2);
                sum1v    += cpowi * sums2[i] / (i+2);
                sum2v    += cpowi * sums2[i];
                if(j==c2){
                  cpow = (1 - cpowi * cvc * cvc)/(i+3);
                  sum1d2vdx2 += cpow * sums4[i] / (i+2);
                  sum2d2vdx2 += cpow * sums3[i] * (i+1) / (i+2);
                  sum3d2vdx2 += cpow * sums2[i] * (i+1) / (i+2);
                  sum4d2vdx2 += cpow * sums2[i] * (i+1);
                  sum5d2vdx2 += cpow * sums1[i] * (i+1) / (i+2);
                  sum6d2vdx2 += cpow * sums1[i] * (i+1);
                  sum7d2vdx2 += cpow * sums1[i] * (i+1) * (i+3);
                }
                 }
            P_dvdx         = cvc * (sqrpi * hh * sum1dvdx + 2 *
                             pi * dd * sum2dvdx + d2 * sum3dvdx - dd * x1 *
                             sum4dvdx);
            Iy_udvdx[ff] += nor * sumu * hh * P_dvdx;
            P_v            = -sum1dhdt * dd - sum2dhdt * pi * hh;
            Iy_dvdt[ff]  += nor * cvc * hh * (P_v * (pi * sum2dvdx
                             - x1 * sum4dvdx) + (-2.0 * sum2dhdt * pi * dd +
                             sum3dhdt * pi * pi * hh - sum1dhdt * d2) * sum3dv
dx);
            if(j==c2){
              Iy_d2vdx2 = hh * (sum1d2vdx2 * sqrpi * pi * hh - 3 *
                          sum2d2vdx2 * sqrpi * dd + 3 * sum3d2vdx2 * pi * d2
-
                          3 * sum4d2vdx2 * pi * dd * x1 + sum5d2vdx2 * d3 - 3
 *
                          sum6d2vdx2 * d2 * x1 + sum7d2vdx2 * dd * x1 * x1);
            }
            cvc -= delc;
            if(j==0) qc=2;
         }

         // ***  p1-p0  ***

         p0_p1 = prss[ff];

         if(ff==0){
           p4_p1 = p4_p0 + p0_p1;}
         else
             p4_p1 = -prss[ff-1] + p0_p1;

         p2_p3 = Ix_d2u_dy2 + sqraver * Ix_d2u_dx2 - re * aver * (Ix_dudt +
                 Ix_dudx + Ix_vdudy) + c[g] * sqraver * ( Ix_dhdx_d2vdy2 +
                 sqraver * Ix_dhdx_d2vdx2 - re * aver * (Ix_dhdx_dvdt +
                 Ix_dhdx_udvdx + Ix_dhdx_vdvdy));

        //--- calculation of p2-p1 ---
        Iy_d2v_dy2 = funct_dvdy(x[ff],1,hh,dd)
                      - funct_dvdy(x[ff],c[g],hh,dd);
        coprex= Sqr(funct_v(x[ff],1,hh,dd));
        Iy_vdvdy=funct_v(x[ff],c[g],hh,dd);
        Iy_vdvdy=0.5*(coprex-Sqr(Iy_vdvdy));

        //*** (p4-p3) ***
        if(ff==0){
          p34 = p3_p4[g];}
        else
            p34 = -p1_p2;

        //*** (p2-p1) ***

        coprex = sqraver * Iy_d2v_dy2 + Sqr(sqraver) * Iy_d2vdx2;
        p1_p2  = -coprex + re*sqraver*aver * (Iy_dvdt[ff] +
                 Iy_udvdx[ff] + Iy_vdvdy);

        rd = p1_p2 + p2_p3 + p34 + p4_p1;

            //calculate the alfa elements
        for(k=0;k<nkc;k++){
           cpowi = 1.0;
           for(i=0;i<nic;i++){

              cpowi *= c[g];
           // Ix_dudt
              if(ff==0){
              alfap_1 = re * aver * cpowi * (OddNeg(k) + cosk[k]) / (pi*(k+1)
);}
              else
              alfap_1 = re * aver * cpowi * (-cos(pi*(k+1)*x[ff-1]) + cosk[k]
) /
                        (pi*(k+1));
           // Ix_dhdx_dvdt
              coprex  = (i+1) * ii7[k] - pi * (k+1) * ii8[k];
```

116

```
            alfap_2= re*sqraver*aver*cpowi*c[g]*c[g]
                     * coprex/(i+2.0);
         // Iy_dvdt
            if(ff==0){
              coprex= ((i+1) * sink[k] * dd - pi * (k+1) * hh * cosk[k]) *
                      (-re*sqraver*aver*hh);}
            else
                coprex= ((i+1) * sink[k] * dd - pi * (k+1) * hh * cosk[k])
*
                        (-re*sqraver*aver*hh)
                        -((i+1) * sin(pi*(k+1)*x[ff-1]) * dhdx[ff-1] -
                        pi * (k+1) * hx[ff-1] * cos(pi*(k+1)*x[ff-1])) *
                        (-re*sqraver*aver*hx[ff-1]);
            alfap_3= coprex*(1.0 - cpowi * c[g] * c[g])/((i+2)*(i+3.0));
         // alfa-elements
            if(k<nkc-1) helpa[i * (nkc-1) + k] = alfap_1 + alfap_2 + alfap_
3;
            else{
                h1 = alfap_1 + alfap_2 + alfap_3;
                for(l=0;l<nkc-1;l++){
                   il = i * (nkc-1) + l;
                   helpa[il] += h1 * (l+1) * OddNeg(nkc+l) / nkc;
                   if(fabs(helpa[il])>mx) mx = fabs(helpa[il]);
                }
            }
        }
      }
      mx *= mx;
      for(k=0;k<(nkc-1)*nic;k++){
         bvec[k] += rd * helpa[k] / mx;
         for(i=0;i<nic*(nkc-1);i++)
            ah[k][i] += helpa[k] * helpa[i] / mx;
      }
    }    // end of ff loop
 }       // end of g loop
}

//-------------------------- End of ContourInt ----------------------------
-
```

APPENDIX

IN- AND OUTPUT VARIABLES AND VARIABLES WHICH ARE USED BUT NOT CHANGED.

```
(Program Marangoni_version_2)
* Index #         : 1
* Title           : Marangoni.cpp
* Last Edit       : 28 January 1992
* Copywright      :
* System          : C++
* Description     : Computation of the velocities in a thin liquid layer
                    caused by the Marangoni effect.
* Date 1st issue  : 19 October 1992
* History         : Originally written in Turbo Pascal
* Status          : -
```

List of functions		member of class
1	Sqr	Basics
2	tmplstmstp	MaraVar
3	OddNeg	Data
4	OutStream	InOutFile
5	InStream	InOutFile
6	ReadData	EnterData
7	StartScreen	EnterData
8	StartNew	Init_New_Run
9	BandDAdjustment	Init_New_Run
10	ScrQuest	Init_New_Run
11	IncorrIndex	Init_New_Run
12	EnterIndex	Init_New_Run
13	c_Comp	c_Computation
14	Zero_Calc	c_Computation
15	SxSgrComputation	SxSgr
16	NewNkNi_uc_h	NewNkNi
17	hGradientCalculation	NewNkNi
18	ATimeStep	TimeStep
19	FactorCalculation	TimeStep
20	ContourInt	TimeStep
21	funct_d2hdx2	Functions
22	funct_dvdy	Functions
23	funct_v	Functions
24	Solve_E_WithTangTens_Cond	SolveWTT_Cond
25	MatSolveWithRightWall	SolveWTT_Cond
26	Simq	SiVaD
27	ludcmp	SiVaD
28	lubksb	SiVaD
29	NewUH	New_UH
30	NewDB	New_DB

```
***************************************************************************
*                                                                         *
*                              Definition                                 *
*                                                                         *
***************************************************************************
```

```
 pi              : About 3.14. The circumference of a circle with diameter 1.
 nkmax           : The maximum number of collocation points in the x direction
.
 nimax           : The maximum number of collocation points in the y direction
.
 nkcmax          : The coefficients d(i,k) in the expansion of the (horizontal
)
                   velocity u(x,y) can be entered into a matrix in which the
                   (i,k)-th element is d(i,k). The maximal dimensions of that
                   matrix are nicmax (number of rows) and nkcmax (number of
                   columns). For an actual matrix of dimension nic x nkc the
                   element d(i,k) is denoted in the program d[i*nkc+k].
 nicmax          : See nkcmax.
 ncoefmax        : The maximum number of coefficients in the expansion of the
                   surface height, exclusive of the zeroth coefficient.
                   nkmax < ncoefmax < 2 * nkmax + 2.
 lengt           : The halfwidth of the liquid film.
 part            : The number of integration steps. A good value for part is 5
12.
                   A larger value (e.g. 1024) possibly increases the accuracy
of
                   the calculations at the cost of a larger computational burd
en.
 aver            : The aspect ratio thichn / lengt.
 thickn          : The initial surface height.
 rho             : The liquid mass density.
 mic             : The dynamic liquid viscosity.

REMARK
 nkmax >= nkcmax; nimax >= nicmax; nkmax < ncoefmax < 2 * nkmax + 2;

**************************************************************************
*                                                                        *
*                                 Basics                                 *
*                                                                        *
**************************************************************************

double Sqr(double r);     //Sqr(r) = r * r;
 1

class Point {
public:
      int x;
      int y;
};

IOMANIPdeclare(Point);

ostream& xy(ostream& os, Point p);

OMANIP(Point) xy(int x, int y);

ostream& cleol(ostream& os);
```

```
**************************************************************************
*                                                                        *
*                            class MaraVar                               *
*                                                                        *
**************************************************************************

protected:
   static int    ntimesteps, ni, nk, nic, nkc, Ncoef_b, par, step;
   static double opps1, opps2, delsig, sgr_xis1, time, timestep;
   static double d[nkcmax*nicmax], dudt_old[nkmax*nimax],
                 e_old[nkcmax*nicmax], b[ncoefmax+1], dhdt_old[nkmax],
                 sx[nkmax], sgr[nkmax], dsxdt[nkmax],
                 d2hdxdt_old[nkmax], c[nimax], x[nkmax];
public:
        int stp() { return step; }
        void plusstep() { step++; }
        void tmplstmstp() { time += timestep; }
2
        int ntmstps() { return ntimesteps; }


**************************************************************************
*                                                                        *
*                   class Data : public virtual MaraVar                  *
*                                                                        *
**************************************************************************

protected:
static double
            dhdt[nkmax], d2hdxdt[nkmax], d3hdtdx2[nkmax],
        ah[(nkcmax-1)*nicmax][(nkcmax-1)*nicmax], helpa[(nkcmax-1)*nicmax],
            bvec[nkcmax*nicmax], re, dudy_y0[nkmax], hx[nkmax], uc[nkmax*ni
max],
        dhdx[nkmax], d2hdx2[nkmax], d3hdx3[nkmax], dudt[nkmax*nimax];
public:
        double v[nkmax];
        int OddNeg(int);     // OddNeg(k) = (-1)^k;
3


**************************************************************************
*                                                                        *
*               class InOutFile : public virtual MaraVar                 *
*                                                                        *
**************************************************************************

public:
        int c123;
        char* filename[10];
        void OutStream();
        void InStream();
        InOutFile(){ *filename = "h" }
```

```
void InOutFile::OutStream()
4
---------------------------

INPUT
 --
OUTPUT
                 :  The parameters nk, ni, ntimesteps, nkc, nic, Ncoef_b,
                    opps1, opps2 and time, and the values for the array
                    elements d[0,...,nkc*nic-1] and b[0,...,Ncoef_b] are
                    written out to file MaranOut
UNCHANGED
 --
REMARK
 The meaning of the mentioned parameters are explained in InStream()


void InOutFile::InStream()
5
--------------------------

INPUT
 --
OUTPUT (FILE INPUT)
 nk              :  The number of collocation points in the x direction
 ni              :  The number of collocation points in the y direction
 ntimesteps      :  The number of time steps
 nkc             :  For the coefficients d(i,k) in the expansion of the
                    velocity u 0 <= k < nkc. {So d(i,k) = d[i*nkc+k]}
 nic             :  For the coefficients d(i,k) in the expansion of the
                    velocity u 0 <= i < nic
 Ncoef_b         :  The number of b coeffitients (exclusive of b[0], in the
                    expansion of the liquid height (interface) h(x))
 opps1           :  Surface tension liquid 1
 opps2           :  Surface tension liquid 2
 b               :  See output BAndDAdjustment() (9)
 d               :  See output BAndDAdjustment() (9)
 time            :  Starting time = 0, after x time steps of say 0.01 (s.)
                    time = x * 0.01 (s.)
OUTPUT
 delsig          :  opps1 - opps2.
 c123            :  Integer.
                    if c123 = 0 -> The name given to *filename is not an
                                   existing file;
                    || c123 = 1 -> *filename contains the name of an existing
                                   data file. This file is opened and the
                                   data is read. The information contained
                                   in such a data file is the information
                                   of a file like MaranOut. (See OutStream)
                    fi
filename[10]     :  If the entered data file name does not exist or if that
                    file cannot be opened, *filename is changed into "h".
```

```
                     When *filename remains "h" the data has to be entered by h
and
par                :  if par  = 2 ->  Data input via data file
                      || par != 2 ->  Data input not via data file
                      fi
UNCHANGED
 --


**********************************************************************
*                                                                    *
*               class EnterData : public InOutFile                   *
*                                                                    *
**********************************************************************

private:
        int ready, st, sw, fn;
        double sf;
        void StartScreen();
public:
        void ReadData();


void EnterData::ReadData()
6
--------------------------

INPUT
 --
OUTPUT (SCREEN INPUT)
 nk            :  The number of collocation points in the x direction
                  (default nk = 3)
 ni            :  The number of collocation points in the y direction
                  (default ni = 3)
 ntimesteps    :  The number of time steps
                  (default ntimesteps = 1)
 nkc           :  The number of columns in matrix d
                  (default nkc = 3)
 nic           :  The number of rows in matrix d
                  (default nic = 3)
 Ncoef_b       :  The number of b coeffitients (exclusive of b[0], in the
                  expansion of the liquid height (interface) h(x))
                  (default Ncoef_b = 3)
 opps1         :  Surface tension liquid 1 (see REMARK)
 opps2         :  Surface tension liquid 2 (see REMARK)
 filename[10]  :  Name of the input data file. If *filename = "h", the data
                  has to be entered by hand and so no input file is used.
                  If one enters "x", "x" is changed into "MaranOut", the
                  default input file
OUTPUT
 delsig        :  opps1 - opps2 (see REMARK)
 ready         :  Only for local use.
 st            :  Communication parameter between memberfunctions of EnterDa
ta
```

```
 sw             :  See st. st, sf and sw are used in connection with opps1 an
d
                   opps2
 fn             :  Communication parameter between memberfunctions of EnterDa
ta
                   concerning information on the screen about the possiblies
of
                   entering data
 sf             :  See st
UNCHANGED
 --
CALLED FUNCTIONS
 StartScreen()
 InStream()

REMARK
 delsig: Normally delsig is the difference in surface tension between water
         and alcohol at the interface. The value of delsig is:
         Õ(water) - Õ(alcohol) = 72.3e-3 - 22.8e-3 = 49.5e-3 [N/m].
 Output file "OutData" is opened. The screen input values are stored in Out
Data


void EnterData::StartScreen()
7
-----------------------------

INPUT
 --
OUTPUT
 nk             :  If on entry nk < 1 or nk > nkmax, nk is set equal to nkmax
                   If nk < nkc, nk is set equal to nkc
 ni             :  If on entry ni < 1 or ni > nimax, ni is set equal to nimax
                   If ni < nic, ni is set equal to nic
 nkc            :  If on entry nkc < 1 or nkc > nkcmax, nkc is set equal to
                   nkcmax. If nkc > nk, nk is set equal to nkc
 nic            :  If on entry nic < 1 or nic > nicmax, nic is set equal to
                   nicmax. If nic > ni, ni is set equal to nic
 Ncoef_b        :  If on entry Ncoef_b < 1 or Ncoef_b > 2 * nk + 1, Ncoef_b i
s
                   set equal to 2 * nk + 1

UNCHANGED
 ntimesteps, opps1, opps2

REMARK
 For the meaning of the output parameters see ReadData()


**************************************************************************
*                                                                        *
*         class Init_New_Run : public c_Computation, public SxSgr        *
*                                                                        *
**************************************************************************
```

```
private:
    int n1,n3,k,tick,ready;
    char ch;
    double sgr_xis1;
    void IncorrIndex();
    void EnterIndex();
    void ScrnQuest(char);
    void BandDAdjustment();
public:
        void StartNew();


void Init_New_Run::StartNew()
8
-----------------------------

INPUT
 --
OUTPUT
 b[0]          :  b[0] = 1. Initialisation of the first element of vector b
 sx            :  see output SxSgr::SxSgrComputation()
 sgr           :  see output SxSgr::SxSgrComputation()
 dsxdt         :  see output SxSgr::SxSgrComputation()
 sgr_xis1      :  see output SxSgr::SxSgrComputation()
 c             :  array [0,.....,ni-1] of the ni collocation points i.e.
                  the zeros of Legendre polynomals in the c direction.
 x             :  array [0,.....,nk-1] of the nk collocation points i.e.
                  zeros of Legendre polynomals in the x direction.
 b             :  see output BandDAdjustment()
 d             :  see output BandDAdjustment()

UNCHANGED
 ni, nk, delsig, lengt, opps1, time

CALLED FUNCTIONS
 c_Comp(double*,int)
 SxSgrComputation()
 BandDAdjustment()


void Init_New_Run::BandDAdjustment()
9
------------------------------------

INPUT
 --
OUTPUT (SCREEN INPUT)
 b             :  array[0,.....,Ncoef_b] of the coefficients of the expansio
n
                  of the height h(x) of the interface at time 0.
 d             :  array[0,.....,nk*ni-1] of the coefficients of the expansio
n
                  of the velocity in x-direction u at time 0.
 par            :  par = 1 (this means data is entered by hand)
 time           :  if time = 0 -> The program starts at time 0
                   || time = 2 -> The program starts at time 2
                                 (The idea behind this is that at time = 2
                                  there is not a driving force (see
                                  SxSgrComputation (15)))
                   fi
UNCHANGED
 --
CALLED FUNCTIONS
 ScrQuest(char*)
 IncorrIndex()
 EnterIndex()
 BandDAdjustment()

REMARK
 Only the elements b[1], b[2], ... , b[Ncoef_b-1] can be entered because
 b[0] = 1 (by definition) and b[Ncoef_b] is computed using the fact that
 h(1) = 1.
 Not all elements of the ni x nk matrix d can be entered. All elements can
 be entered except the elements of the last row and those of the last colum
n.
 This is due to the fact that the d-coefficients have to fullfill the
 tangential stress condition and the no-slip condition of the right wall


void Init_New_Run::ScrQuest(char* a)
10
------------------------------------

INPUT
 a              :  The function puts on the screen the text 'Do you want
                   to change the ... coefficients (y/n/a)?'
                   with the value of character a at the place of the dots.
OUTPUT
 tick           :  If the answer to the preceding question is a then
                   tick = 2 else tick = 1
UNCHANGED
 --


void Init_New_Run::IncorrIndex()
11
--------------------------------

INPUT
 --
OUTPUT (depending on SCREEN INPUT)
 tick           :  if tick = 0 -> After entering a data point the computer
                                  asks ... Ready (y/n)?
                   || tick = 1 -> The text Ready (y/n)? does not appear on
                                  the screen. (See at tick = 0).
                   || tick = 2 -> The computer shows all indices one at a tim
e.
```

```
                                    You only need to enter the corresponding da
ta.
                fi
ready          :   Stop entering data: ready = 1
UNCHANGED
 --


void Init_New_Run::EnterIndex()
12
-------------------------------

INPUT
 --
OUTPUT (SCREEN INPUT)
 k             :   The new index
OUTPUT
 tick          :   see IncorrIndex()
UNCHANGED
 --


*************************************************************************
*                                                                       *
*               class c_Computation : public virtual MaraVar            *
*                                                                       *
*************************************************************************

private:
        void Zero_Calc(double*,int);
public:
        void c_Comp(double*,int);


void c_Computation::c_Comp(double *c, int jx)
13
---------------------------------------------

INPUT
 jx            :   The number of collocation points.
OUTPUT
 c             :   array [0,.....,jx-1] of the jx-collocation points i.e.
                   the zeros of the Legendre polynomal of order jx.
                   c[0] > c[1] > ... > c[jx-1]
UNCHANGED
 --
CALLED FUNCTIONS
 Zero_Calc(double*,int)

REMARK
 The Legendre polynomials are normally defined on the interval [-1,1].
 In this function the Legendre polynomials are defined on the interval [0,1
].
```

```
void c_Computation::Zero_Calc(double *c, int jx)
14
------------------------------------------------

INPUT
 jx          :  The number of collocation points.
OUTPUT
 c           :  array [0,.....,jx-1] of the jx-collocation points i.e.
                the zeros of the Legendre polynomal of order jx.
UNCHANGED
 --
CALLED FUNCTIONS
 --
REMARK
 The Legendre polynomials are normally defined on the interval [-1,1].
 In this function the Legendre polynomials are defined on the interval [0,1
].


*************************************************************************
*                                                                       *
*                 class SxSgr : public virtual MaraVar                  *
*                                                                       *
*************************************************************************

public:
      void SxSgrComputation();


void SxSgr::SxSgrComputation()
15
------------------------------

INPUT
 --
OUTPUT
 sx          :  array [0,.....,nk-1] with the values of the tangential
                surface tensions at the interface depending on the
                x coordinates.
 sgr         :  array [0,.....,nk-1] with the values of the normal surface
                tensions at the interface depending on the x coordinates.
 dsxdt       :  array [0,.....,nk-1] with the values of the normal surface
                tension at the interface differentiated in time depending
                on the x-coordinates.
 sgr_xis1    :  value of the normal surface tension at the interface at
                x coordinate x = 1.
UNCHANGED
 delsig, x, nk, lengt, opps1, time


*************************************************************************
*                                                                       *
*                        class Contour_Int_New                          *
*                                                                        *
**************************************************************************

private:
 int m;
 double xl;

public:
       double prss[nkmax];
       double dudx[nkmax];


**************************************************************************
*                                                                        *
*            class NewNkNi : public SxSgr, public c_Computation,         *
*            public hGradient                                            *
*                                                                        *
**************************************************************************

public:
      void NewNkNi_uc_h()

void NewNkNi::NewNkNi_uc_h()
16
----------------------------

INPUT
 --
OUTPUT
 re              :  Reynolds-number.
 d               :  array[0,.....,nk*ni-1] of the coefficients of the expansio
n
                    of the velocity in x-direction u at time 0
 uc              :  array[0,.....,nk*ni-1] The velocity in x-direction at
                    the interface
 helpa           :  array[0,.....,ni*(nkc-1)-1] Help array
 h               :  see output hGradientCalculation()
 dhdx            :  see output hGradientCalculation()
 d2hdx2          :  see output hGradientCalculation()
 d3hdx3          :  see output hGradientCalculation()
UNCHANGED
 sx,delsig,aver,thickn,rho,mic,ni,nk,nic,nkc,bvec,ah

CALLED FUNCTIONS
 hGradientCalculation()
 MatSolveWithRightWall()
 OddNeg(int)
 Sqr()

REMARK
 The d-coefficients are calculated again using the following equations:
 1. The tangential stress condition
 2. The values for the d-coefficients
 Note that when par=1 (i.e. when the d-values are entered by hand, and
 time=0) not all d-values are known


void NewNkNi::hGradientCalculation()
17
------------------------------------

INPUT
 --
OUTPUT
 h              :  array[0,.....,nk-1] of the height of the interface, depend
ing
                   on x.
 dhdx           :  array[0,.....,nk-1] of the first derivative of the height
of
                   the interface.
 d2hdx2         :  array[0,.....,nk-1] of the second derivative of the height
 of
                   the interface.
 d3hdx3         :  array[0,.....,nk-1] of the third derivative of the height
of
                   the interface.
UNCHANGED
 x,b,nk,Ncoef_b


***************************************************************************
*                                                                         *
*  class TimeStep : public Contour_Int_New, public New_DB, public SxSgr   *
*                                                                         *
***************************************************************************

private:
        void FactorCalculation_New();
        void FactorCalculation();
        void ContourInt();
        void NewUH();
public:
      void ATimeStep();


void TimeStep::ATimeStep()
18
--------------------------

INPUT
 --
OUTPUT
                :  see output CALLED FUNCTIONS
UNCHANGED
 see Called Functions

CALLED FUNCTIONS
 FactorCalculation()
 ContourInt()
 Solve_E_WithTangTensCond()
 NewUH()
 SxSgrComputation()
 NewDB()

REMARK
 The time and the b- and d-values at this time are appended
 to output file OutData


void TimeStep::FactorCalculation()
19
----------------------------------

INPUT
 --
OUTPUT
 v              : array [0,.....,nk-1] The velocity component in the y
                  direction at the interface
 dudx           : array [0,.....,nk-1] The x-differentiated velocity
                  component u at the points (x[g],h(x[g])), g=0,...,nk-1.
 d2hdxdt        : array [0,.....,nk-1] The liquid height h differentiated
                  to x and t.
 d3hdtdx2       : array [0,.....,nk-1] The height h once time differentiated
                  and twice x differentiated.
 dhdt           : array [0,.....,nk-1] The time differentiated liquid height
                  h at the nk collocation points in the x direction.
 prss           : array [0,.....,nk-1] Difference in normal stress at the
                  interface (p-p0) for nk x values.
UNCHANGED
aver,nk,ni,pi,x,d,dhdx,hx,d2hdx2,d3hdx3,uc,sgr,sx


void TimeStep::ContourInt()
20
---------------------------

INPUT
 --
OUTPUT
 helpa          :  array [0,.....,nic*(nkc-1)-1] A help array for matrix ah
                   and vector bvec.
 bvec           :  array [0,.....,nic*(nkc-1)-1] Right hand side of the matri
x
                   equation ah.e = bvec. Contribution of the contour integral
s
                   to vector bvec. e is the time differentiated b-vector.
 ah             :  array [0,.....,nic*(nkc-1)-1][0,.....,nic*(nkc-)-1] The
                   matrix in the equation ah.e = bvec. Considered are only th
e
                   contributions of the contour integrals to the matrix ah.
UNCHANGED
 aver, re, pi, sgr_xis1, ni, nk, c, x, b, d, Ncoef_b, prss
```

```
CALLED FUNCTIONS
 OddNeg(int)
 funct_v(double,double,double,double)
 funct_dvdy(double,double,double,double)
 funct_d2hdx2(double)


*************************************************************************
*                                                                       *
*                class Functions : public virtual Data                  *
*                                                                       *
*************************************************************************

 public:
     double funct_d2hdx2(double);
     double funct_dvdy(double,double,double,double);
     double funct_v(double,double,double,double);


double Functions::funct_d2hdx2(double xi)
21
-----------------------------------------

INPUT
 xi             : x value.
OUTPUT
                : The second derivative of the liquid height function h
                  (=d2hdx2) at x = xi.
UNCHANGED
 Ncoef_b,b,pi


22
double Functions::funct_dvdy(double xi ,double ci, double hxi, double dhdxi
)
---------------------------------------------------------------------------
-

INPUT
 xi             : x value.
 ci             : y value divided by the liquid height at x = xi.
 hxi            : The liquid height at x = xi.
 dhdxi          : The derivative of the liquid height at x = xi.
OUTPUT
                : dv/dy at (x,y) = (xi,ci*hxi).
UNCHANGED
 pi,nk,ni,d

double Functions::funct_v(double xi ,double ci, double hxi, double dhdxi)
23
---------------------------------------------------------------------------
-
```

```
INPUT
 xi             : x value.
 ci             : y value divided by the liquid height at x = xi.
 hxi            : The liquid height at x = xi.
 dhdxi          : The derivative of the liquid height at x = xi.
OUTPUT
                : The vertical velocity component v at (x,y) = (xi,ci*hxi).
UNCHANGED
 pi,nk,ni,d


*******************************************************************
*                                                                 *
*       class SolveWTT_Cond : public virtual Data, public SiVaD   *
*                                                                 *
*******************************************************************

public:
       void MatSolveWithRightWall();
       void Solve_E_WithTangTens_Cond();


void SolveWTT_Cond::Solve_E_WithTangTens_Cond()
24
-----------------------------------------------

INPUT
 --
OUTPUT
 helpa          : array [0,.....,nic*(nkc-1)-1] A help array for matrix ah
                  and vector bvec.
 bvec           : array [0,.....,nic*(nkc-1)-1] The time differentiated
                  coefficients d(i,k).
 ah             : array [0,.....,nic*(nkc-1)-1][0,.....,nic*(nkc-1)-1]
                  An array of zeros.
UNCHANGED
 aver,nk,ni,pi,x,hx,dhdt,dhdx,d2hdx2,d2hdxdt,dsxdt,sgr,sx

CALLED FUNCTIONS
 OddNeg(int)
 Sqr(double)
 MatSolveWithRightWall()

REMARK
 In this routine cross differentiation of the Navier-Stokes equations
 is applied to equal the velocities at the bottom to about zero. The
 time differentiated tangential stress condition is used too. Finally
 the time differentiated coefficients of u are computed as solution
 of the equation ah.x = bvec.

void SolveWTT_Cond::MatSolveWithRightWall()
25
--------------------------------------------
```

```
INPUT
 --
OUTPUT
 bvec            : array [0,.....,nk-1] This vector contains the time
                   differentiated coefficients d(i,k) in the expansion
                   of the tangential velocity component u.
 ah              : array [0,.....,nic*(nkc-1)-1][0,.....,nic*(nkc-1)-1]
                   An array of zeros.
UNCHANGED
 nk,ni,ah

CALLED FUNCTIONS
// ludcmp(int,int*,double)
// lubksb(int,int*,double*)
 Simq(double*,int)

***************************************************************************
*                                                                         *
*                     class SiVaD :public virtual Data                   *
*                                                                         *
***************************************************************************

public:
      void Simq(double*,int);
      void lubksb(int,int*,double*);
      void ludcmp(int,int*,double*);
      void SiVaDe(double*,double*,double*,int,int);


void SiVaD::Simq(double* r,int i)
26
---------------------------------

INPUT
 r             :  The right vector in the matrix equation ah x = r
 i             :  The dimension of the square matrix ah
OUTPUT
 r             :  The matrix equation ah(transpose) x = r is solved,
                  ah and r are known and x has to be computed. On
                  output the value of x is written in vector r. Thus
                  the right vector is overwritten. See further REMARK
 ah            :  Not important
UNCHANGED
 --
REMARK
 The matrix ah is symmetric, so that there is no difference between
 ah(transpose) and ah


(deleted)
void SiVaD::ludcmp(int n,int* indx,double &p)
27
---------------------------------------------
```

```
INPUT
 n             :  dimension of array ah
OUTPUT
 indx          :  array [0,.....,r-1] with r >= n. Help array of type intege
r
                  for routine lubksb.
 p             :  Help variable.
UNCHANGED
 --

(deleted)
void SiVaD::lubksb(int n,int* indx,double* z)
28
---------------------------------------------

INPUT
 n             :  dimension of array ah
 z             :  array [0,.....,r-1] with r >= n. Right-hand side of the
                  equation ah.x = z
OUTPUT
 indx          :  array [0,.....,r-1] with r >= n. Help array of type intege
r.
 z             :  array [0,.....,r-1] with r >= n. The solution vector.
UNCHANGED
 --
REMARK
 The routines ludcmp and lubksb are described in Numerical Recipes.


***************************************************************************
*                                                                         *
*                 class New_UH : public virtual Data                      *
*                                                                         *
***************************************************************************

public:
      void NewUH();


void New_UH::NewUH()
29
--------------------

INPUT
 --
OUTPUT
 bvec          :  array [0,.....,nkc*nic-1] The time differentiated
                  coefficients d(i,k)
 e_old         :  array [0,.....,nkc*nic-1] The time differentiated
                  coefficients d(i,k)
 dudt          :  array [0,.....,ni*nk-1] The time differentiated velocity u
 dudt_old      :  array [0,.....,ni*nk-1] The time differentiated velocity u

 dhdt_old      :  array [0,.....,nk-1] The time differentiated height h
 d2hdxdt_old    :  array [0,.....,nk-1] The liquid height h differentiated
                   to x and t
 uc             :  array[0,.....,nk*ni-1] The velocity in x-direction at
                   the coordinates of the collocation-points
 hx             :  array [0,....,nk-1] The height h
 dhdx           :  array [0,....,nk-1] The height h diffentiated to x
 d              :  array [0,.....,nkc*nic-1] The velocity coefficients d(i,k)
 time           :  A timestep has been added to variable time
UNCHANGED
 ni,nk,nic,nkc,c,dhdt,step,timestep,d2hdxdt

CALLED FUNCTIONS
 OddNeg(int)
 tmplstmstp()

REMARK
 In this routine time is raised by a timestep. New values for e.g. dudt
 at this new time is computed using the Adams-Bashford formula.
 E.g. uc = uc + (1.5 * dudt - 0.5 dudt_old) * timestep.
 dudt_old is the time differentiated velocity u (=uc) of the previous time
 and dudt of the new time. After this dudt_old = dudt (i.e. The dudt now
 is the dudt_old for the next visit of this routine


**************************************************************************
*                                                                        *
*      class New_DB : public c_Computation, public SolveWTT_Cond,        *
*                     public virtual Functions                           *
*                                                                        *
**************************************************************************

private:
       void MatSolveForVector0();
public:
      void NewDB();


void New_DB::NewDB()
30
--------------------

INPUT
 --
OUTPUT
 helpa          :  A help array for matrix ah and vector bvec
 bvec           :  A help array. On exit bvec is an array of zeros
 ah             :  The matrix in the equation ah.x = bvec. On entry and on ex
it

                   an array of zeros
 b              :  array[0,.....,Ncoef_b] of the coefficients of the expansio
n

                   of the height h(x) of the interface, with b[0] = 1.
 d              :   array[0,.....,nic*nkc-1]. The expansion coefficients of u
                   (=uh). The d values are calculated once at most in this
```

```
                        routine during a program run. (See REMARK)
 par                 :  par=0. It isn't important any more for the program
                        to distinguish between the different forms of entering the
                        starting data
UNCHANGED
 nk,ni,Ncoef_b,nic,nkc,x,c,pi,hx,dhdx,lengt,aver,sx

CALLED FUNCTIONS
 MatSolveForVector0()
 OddNeg()
 Sqr()

REMARK
 The formulae for the expansion of the liquid height h (=hx) and for its
 derivative dhdx and the values for hx and dhdx as computed in NewUH
 provide a set of equations from which the new b-coefficients are calculate
d.
 If the program starts from scratch, i.e at time = 0 and with all b- and
 d-coefficients zero (except b[0]=1) then in the first visit of this routin
e
 the d-coefficients are calculated using a set of linear equations defined
by:
 1. At all collocation point (except the collocation points with highest
 y-coordinate) define u=0. 2. The tangential stress condition at the
 interface.
```

Description of the structure of the algorithm Marangoni.cpp.
The description is in chronological order.

```
* Index #         : 2
* Title           : Marangoni.cpp
* Last Edit       : Februari 5 1993
* Copywright      : Nieuwenhuizen
* System          : C++
* Description     : Computation of the velocities in a thin liquid layer
                    caused by the Marangoni effect.
* Date 1st issue  : October 19 1993
* History         : Originally written in Turbo Pascal
* Status          : -

1 EnterData::ReadData             Routine to enter screen input data.

  a. EnterData::StartScreen       Shows the default parameters and
                                  offers the possibility to change them.


  b. InOutStream::InStream        Data input via data file. Whether the
                                  data is entered by hand or with the
                                  help of a data file has been decided
                                  in the routine StartScreen.

2 Init_New_Run::StartNew          Define b[0]=1.

  a. c_Computation::c_Comp        This routine is called twice. First to
                                  compute the collocation points in the
                                  x direction, and second to compute the
                                  collocation points in the y direction.

  b. SxSgr::SxSgrComputation      Computes at the interface
                                           sx (i.e. S1), its time-derivativ
e
                                           dsxdt,
                                           sgr (The surface tension gradien
ts),
                                           sgr_xis1 (i.e. Sgr at the wall)

3 Init_New_Run::BandDAdjustment   .Screen input of the coefficients b[.]
                                  of the expansion h of the surface interfa
ce.
                                  .Screen input of the coefficients d[.]
                                  of the expansion u of the liquid velociti
es
                                  in the x direction.
                                  .Asks wether there is a driving force.
                                  If the answer is no, the starting time is
                                  set equal to 2 (s.). At that time sx=0
                                  and sgr is constant. (See definition sx a
nd
                                  sgr in routine SxSgrComputation).
  a. ScrQuest                     This routine puts on the screen the quest
ion

                                  if you want to change the b or d
                                  coefficients.

  b. EnterIndex                   Provides the b and d coefficients
                                  automatically with indices.

  c. IncorrIndex                  Gives a beep when you enter an incorrect
                                  index.

step = 1;
WHILE ( STEP <= TOTAL NUMBER OF TIME STEPS )
{
  step = step + 1;

  4 RunScr::RunScreen               Shows during runtime at the top of the
                                    screen information about some input
                                    parameters.

  5 NewNkNi::NewNkNi_uh_c           .Computes re (A Reynolds number).
                                    .The tangential stress condition togeth
er

                                    with the entered or calculated d values
                                    form a set of equations from which new
                                    d values are calculated.
                                    .Computes uc (i.e. the expansion u of t
he

                                    liquid velocities in the x direction at

                                    the interface).

    a. hGradient::hGradientCalculation
                                    Computes the height of the liquid surfa
ce h

                                    hx as well as its derivatives
                                    dhdx
                                    d2hdx2
                                    d3hdx3

  6 TimeStep::ATimeStep

    a. TimeStep::FactorCalculation  Computes v (the normal velocities of th
e

                                              liquid at the interface).
                                           dudx (the x-derivative of the
                                                 tangential velocities of
                                                 the liquid at the interf
ace)

                                           d2hdxdt
                                           d3hdtdx2
                                           dhdt
                                           prss (part of the contour
                                                 integrals used in routin
```

e

ContourInt).

b. TimeStep::ContourInt .Computation of the contour integrals a
s
described in the theory.
The order in which the contour integral
s
are computed is as follows:
The first contour integral starts in th
e
collocation point with the heighest
coefficients. The second contour integr
al
starts at the collocation point next to
the first collocation point and with th
e
same distance from the bottom. The last
contour integral starts at a collocatio
n
point which has the smallest coordinate
s
of all collocation points.
Most of the integrals are computed
numerically with Simpson's rule.

c. SolveWTT_Cond:: Solve_E_WithTangTens_Cond
.The time derivated tangential stress
condition gives a set of equations
in the variables e[.] (e[.] is the
time derivative of d.]).
.Differentiation of the Navier Stokes
equations to x and y and equating
the pressure terms gives a set of
equations in e[.] defined at the bottom
.
.Together with the equations computed
in the routine ContoutInt the e values
are computed.

d. TimeStep::NewUH A new time. (time = time + timestep)
The following arrays are computed using
the Adams-Bashford method:
hx (the height of the liquid surface)
dhdx (the derivative of hx)
d

e. SxSgr::SxSgrComputation Computes at the interface
sx (i.e. S1), its time-derivat
ive
dsxdt,
sgr (The surface tension
gradients),
sgr_xis1 (i.e. Sgr at the wall

```
)

    f. New_DB::NewDB                    Computes the new b values using the val
ues
                                        of hx and dhdx which have been computed
 in
                                        routine NewUH.

                                        Writing the time, the b and d values to
                                        the data file outdata.
} // END OF LOOP

7                                     Writing the parameters and b and d values

                                      to data file MaranOut.

// end of program
```

APPENDIX

0.1 Formulae

$$
\begin{aligned}
y &= h \cdot c \\
u(x,y) &= \sum_{i=0}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} d_{i,k} \cdot \sin(\pi(k+1)x) \cdot \left(\frac{y}{h}\right)^{i+1} \\
d_{i,\mathtt{nkc}-1} &= \frac{1}{\mathtt{nkc}} \cdot \sum_{k=0}^{\mathtt{nkc}-2} (k+1) \cdot d_{i,k} \cdot (-1)^{\mathtt{nkc}+k} \\
h(x) &= \sum_{m=0}^{\mathtt{Ncoefb}} b_m \cdot \cos(\pi m \cdot x) \\
b_0 &= 1 \\
b_{\mathtt{Ncoefb}} &= \sum_{m=1}^{\mathtt{Ncoefb}-1} b_m \cdot (-1)^{m+1+\mathtt{Ncoefb}} \\
\frac{\partial h}{\partial t} &= v - u \cdot \frac{\partial h}{\partial x} \\
e_{i,k} &= \frac{\partial d_{i,k}}{\partial t} \\
\frac{\partial u}{\partial x}(x,y) &= \sum_{i=0}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} d_{i,k} \cdot \cos(\pi(k+1)x) \cdot \pi \cdot (k+1) \cdot \left(\frac{y}{h}\right)^{i+1} - \\
&\quad - \sum_{i=0}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} d_{i,k} \cdot \sin(\pi(k+1)x) \cdot (i+1) \cdot \frac{1}{h} \cdot \frac{\partial h}{\partial x} \cdot \left(\frac{y}{h}\right)^{i+1} \\
\frac{\partial^2 u}{\partial x^2}(x,y) &= \sum_{i=0}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} -d_{i,k} \cdot \sin(\pi(k+1)x) \cdot \pi^2 \cdot (k+1)^2 \cdot \left(\frac{y}{h}\right)^{i+1} - \\
&\quad - 2 \sum_{i=0}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} d_{i,k} \cdot \cos(\pi(k+1)x) \cdot \pi \cdot (k+1) \cdot (i+1) \cdot \frac{1}{h} \cdot \frac{\partial h}{\partial x} \cdot \left(\frac{y}{h}\right)^{i+1} + \\
&\quad + \sum_{i=0}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} d_{i,k} \cdot \sin(\pi(k+1)x) \cdot (i+1) \cdot (i+2) \cdot \left(\frac{\partial h}{\partial x} \cdot \frac{1}{h}\right)^2 \cdot \left(\frac{y}{h}\right)^{i+1} - \\
&\quad - \sum_{i=0}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} d_{i,k} \cdot \sin(\pi(k+1)x) \cdot (i+1) \cdot \frac{\partial^2 h}{\partial x^2} \cdot \frac{1}{h} \cdot \left(\frac{y}{h}\right)^{i+1} \\
\frac{\partial u}{\partial y}(x,y) &= \sum_{i=0}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} d_{i,k} \cdot \sin(\pi(k+1)x) \cdot \frac{i+1}{h} \cdot \left(\frac{y}{h}\right)^{i} \\
\frac{\partial^2 u}{\partial y^2}(x,y) &= \sum_{i=1}^{\mathtt{nic}-1} \sum_{k=0}^{\mathtt{nkc}-1} d_{i,k} \cdot \sin(\pi(k+1)x) \cdot \frac{(i+1) \cdot i}{h^2} \cdot \left(\frac{y}{h}\right)^{i-1}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial u}{\partial t}(x,y) &= \sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} e_{i,k}\cdot\sin(\pi(k+1)x)\cdot\left(\frac{y}{h}\right)^{i+1} - \\
&\quad - \frac{\partial h}{\partial t}\cdot\sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\sin(\pi(k+1)x)\cdot\frac{i+1}{h}\cdot\left(\frac{y}{h}\right)^{i+1} \\
v(x,y) &= \sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\sin(\pi(k+1)x)\cdot\frac{i+1}{i+2}\cdot\frac{\partial h}{\partial x}\cdot\left(\frac{y}{h}\right)^{i+2} - \\
&\quad - \sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\cos(\pi(k+1)x)\cdot\frac{\pi\cdot(k+1)}{i+2}\cdot h\cdot\left(\frac{y}{h}\right)^{i+2} \\
\frac{\partial v}{\partial x}(x,y) &= \sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\sin(\pi(k+1)x)\cdot\frac{\pi^2\cdot(k+1)^2}{i+2}\cdot h\cdot\left(\frac{y}{h}\right)^{i+2} + \\
&\quad + 2\sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\cos(\pi(k+1)x)\cdot\frac{\pi\cdot(k+1)\cdot(i+1)}{i+2}\cdot\frac{\partial h}{\partial x}\cdot\left(\frac{y}{h}\right)^{i+2} + \\
&\quad + \cdot\sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\sin(\pi(k+1)x)\cdot\frac{i+1}{i+2}\cdot\frac{\partial^2 h}{\partial x^2}\cdot\left(\frac{y}{h}\right)^{i+2} - \\
&\quad - \sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\sin(\pi(k+1)x)\cdot\frac{i+1}{h}\cdot\left(\frac{\partial h}{\partial x}\right)^2\cdot\left(\frac{y}{h}\right)^{i+2} \\
\frac{\partial^2 v}{\partial x^2}(x,y) &= \sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\cos(\pi(k+1)x)\cdot\pi^3\cdot(k+1)^3\cdot\frac{h}{i+2}\cdot\left(\frac{y}{h}\right)^{i+2} - \\
&\quad - 3\sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\sin(\pi(k+1)x)\cdot(i+1)\cdot\frac{\pi^2(k+1)^2}{i+2}\cdot\frac{\partial h}{\partial x}\cdot\left(\frac{y}{h}\right)^{i+2} + \\
&\quad + 3\sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\cos(\pi(k+1)x)\cdot(i+1)\cdot\frac{\pi\cdot(k+1)}{i+2}\cdot\frac{\partial^2 h}{\partial x^2}\cdot\left(\frac{y}{h}\right)^{i+2} - \\
&\quad - 3\sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\cos(\pi(k+1)x)\cdot\pi\cdot(k+1)\cdot\frac{i+1}{h}\cdot\left(\frac{\partial h}{\partial x}\right)^2\cdot\left(\frac{y}{h}\right)^{i+2} + \\
&\quad + \sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\sin(\pi(k+1)x)\cdot\frac{i+1}{i+2}\cdot\frac{\partial^3 h}{\partial x^3}\cdot\left(\frac{y}{h}\right)^{i+2} - \\
&\quad - 3\sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\sin(\pi(k+1)x)\cdot\frac{i+1}{h}\cdot\frac{\partial}{\partial x}\cdot\frac{\partial^2 h}{\partial x^2}\cdot\left(\frac{y}{h}\right)^{i+2} + \\
&\quad + \sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} d_{i,k}\cdot\sin(\pi(k+1)x)\cdot(i+1)\cdot(i+3)\cdot\frac{\partial h}{\partial x}\cdot\left(\frac{\partial h}{\partial x}\cdot\frac{1}{h}\right)^2\cdot\left(\frac{y}{h}\right)^{i+2} \\
\frac{\partial v}{\partial y} &= -\frac{\partial u}{\partial x} \\
\frac{\partial^2 v}{\partial y^2}(x,y) &= \sum_{i=0}^{\texttt{nic}-1}\sum_{k=0}^{\texttt{nkc}-1} -d_{i,k}\cdot\cos(\pi(k+1)x)\cdot\pi\cdot(k+1)\cdot\frac{i+1}{h}\cdot\left(\frac{y}{h}\right)^{i} +
\end{aligned}
$$

$$
\begin{aligned}
&+ \sum_{i=0}^{\mathrm{nic}-1} \sum_{k=0}^{\mathrm{nkc}-1} d_{i,k} \cdot \sin(\pi(k+1)x) \cdot \frac{\partial h}{\partial x} \cdot \left(\frac{i+1}{h}\right)^2 \cdot \left(\frac{y}{h}\right)^i \\
\frac{\partial v}{\partial t}(x,y) \quad = \quad & \sum_{i=0}^{\mathrm{nic}-1} \sum_{k=0}^{\mathrm{nkc}-1} e_{i,k} \cdot \sin(\pi(k+1)x) \cdot \frac{i+1}{i+2} \cdot \frac{\partial h}{\partial x} \cdot \left(\frac{y}{h}\right)^{i+2} - \\
&- \sum_{i=0}^{\mathrm{nic}-1} \sum_{k=0}^{\mathrm{nkc}-1} e_{i,k} \cdot \cos(\pi(k+1)x) \cdot \frac{\pi \cdot (k+1)}{i+2} \cdot h \cdot \left(\frac{y}{h}\right)^{i+2} + \\
&+ \frac{\partial h}{\partial t} \cdot \sum_{i=0}^{\mathrm{nic}-1} \sum_{k=0}^{\mathrm{nkc}-1} d_{i,k} \cdot \cos(\pi(k+1)x) \cdot \pi \cdot \frac{(i+1)\cdot(k+1)}{i+2} \cdot \left(\frac{y}{h}\right)^{i+2} + \\
&+ \left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial x} \cdot \frac{\partial h}{\partial x} - u \cdot \frac{\partial^2 h}{\partial x^2}\right) \cdot \sum_{i=0}^{\mathrm{nic}-1} \sum_{k=0}^{\mathrm{nkc}-1} d_{i,k} \cdot \sin(\pi(k+1)x) \cdot \frac{i+1}{i+2} \cdot \left(\frac{y}{h}\right)^{i+2} - \\
&- \frac{\partial h}{\partial t} \cdot \sum_{i=0}^{\mathrm{nic}-1} \sum_{k=0}^{\mathrm{nkc}-1} d_{i,k} \cdot \sin(\pi(k+1)x) \cdot \frac{i+1}{h} \cdot \frac{\partial h}{\partial x} \cdot \left(\frac{y}{h}\right)^{i+2}
\end{aligned}
$$