Save Energy
Part 1: Numerical Method for Heat Conduction In Systems of Arbitrarily Different Materials

New Mexico

Supercomputing Challenge

Final Report

April 1, 2010

Team Number 67
Los Alamos High School

Team Members
Edward Dai
Aidan Bradbury

Teacher: Lee Goodwin
Mentor: William Dai

Executive Summary

In the winter, office buildings are normally heated twenty-four hour a day, although there are no people working after hours. We are going to estimate the amount of energy that can be saved if heating systems operate differently. We plan to do the estimate through numerical simulations for time-dependent heat conduction. As the first part of this two-year project, we have developed a numerical method and a complete set of computer codes for one-, two-, and three-dimensional heat conduction in systems of arbitrarily different materials. Since we are going to solve time-dependent problems, the method is required to be second order accurate. Since the thermal properties of materials involved in the problems may be very different, the steady states can be almost reached for some materials while other materials are still in transient states during certain periods. This requires that the numerical method is able to give correct steady states when time steps are very large. There are two aspects in which the method we have developed is original. First, the method is second order in both space and time when time steps are small and at the same time the method gives correct steady states when time steps are very large. The second original contribution is that we have proposed a correct treatment for systems involving arbitrarily different materials. We have also proposed and implemented an iterative approach to solve the resulting systems of algebraic equations. We have applied the state-of-art multigrid method to our iterative solver, which dramatically reduced the number of iterations and CPU time needed in our computer codes. We have extensively tested the numerical method and computer codes for the features of the method through one-, two-, and three-dimensional numerical examples.

1. Statement of the Problem

In the winter, office buildings are normally heated twenty-four hour a day, although there are no people working after hours. This project is to develop a numerical method and a complete set of computer codes for one-, two-, and three-dimensional time-dependent heat conduction in systems of arbitrarily different materials, so that we could use the set of computer codes to study the heat conduction of office buildings the next year and propose the strategy to save energy. The numerical method to be developed is required to be able to give correct steady states when time steps are very large, because the thermal properties of materials may be dramatically different. The features of the numerical method and computer codes are to be demonstrated through numerical examples.

2. Introduction

Heat conduction is an old topic, and there has been a large amount of literature in scientific and engineering communities, for example, see [2,10] and the references therein. Each method has its advantages depending on the nature of the physical problem to be solved. In terms of accuracy, methods may be divided into first order, second order, and high order ones. If time accuracy is important, the second or higher order of accuracy is preferred. But, normally, a scheme accurate more than the second order is complicated and expensive in CPU time. Therefore, methods with the second order of accuracy have become practical for time-dependent problems.

Methods may also be divided into explicit and implicit methods. An explicit scheme, for example, the forward Euler method, is simple. But the size of time step is limited by a stability condition that is normally much smaller than the required accuracy of physics problems. Therefore, an explicit method is often inefficient for some problems, especially when the thermal diffusivity in a problem varies significantly. On the other hand, in implicit methods the size of time step is not limited by numerical stability conditions, and therefore it may be changed according to the requirement of physics problems. But implicit methods normally involve solving a large set of algebraic equations at each time step.

There are two approaches to solve the large set of algebraic equations arriving from implicit methods, i.e., direct one and iterative one. Direct approach for solving linear algebraic equations are presented in all traditional courses of linear algebra, for example [3]. Generally, exact solvers may not be recommended for two- and three-dimensional problems because of their cost. For iterative solvers, for example [4], important questions are convergence and the rate of convergence.

Two typical implicit methods are the backward Euler method and Crank-Nicolson method [1]. The backward Euler method is first order accurate in time, but numerical errors in the method undergo quick damping for large time steps. Therefore the method is very useful to find steady states. Although Crank-Nicolson method is second order accurate, numerical errors do not damp out for large time steps, and significant numerical errors will be introduced when time steps are very large. This is the reason Crank-Nicolson method cannot be used for steady states.

For systems of multi-materials with significantly different thermal properties, the correct treatment of the discontinuity of the material properties remains to be further investigated. A typical approach is to use the algebraic average of two materials adjacent to a material interface for the flux calculation in numerical methods. This typical approach introduces huge numerical errors when thermal properties of the two materials are very different.

In this paper, we will develop a numerical method for one-, two, and three-dimensional heat conduction problems in systems of multi-materials with arbitrarily different thermal properties. We will derive very simple but correct "effective thermal property" across a material interface for flux calculations. The method is second order accurate in both space and time. The method will give exact steady states when time steps are very large. This property is different from either backward Euler method or Crank-Nicolson method. In addition, we will develop an iterative approach to solve the resulting set of algebraic equations, and apply the multi-grid method to dramatically increase the rate of convergence.

For the method to have the property that leads to exact steady states and the second order accuracy in time, the time discretization to be presented in this paper is adopted from the implicit-explicit hybrid methods for hyperbolic systems of conservation laws [9,12], such as Euler equations, magnetohydrodynamical equation, and radiation hydrodynamics equations.

The plan of the remaining part of this report is as follows. The third section is for the basic equation to be solved. In the fourth section we will present the numerical method, which includes the simultaneous discretization in space and time, realization of the second order of accuracy in time, and consideration of steady states, the correct treatment for the discontinuity of material properties. After that is the section for iterative approach. The multigrid method for the resulting set of algebraic equations is in the sixth section. The seventh section is for numerical examples, which include one-, two-, and three-dimensional cases. The last section is for the conclusion of this report. The appendix contains the source codes of the numerical methods and all the examples in this report.

3. Basic Equations

The heat equation is an important partial differential equation that describes the distribution of heat (or variation in temperature) in a given region over time. For a function T(t, x,y,z) of three spatial variables (x,y,z) and the time variable t, the heat equation is

$$\frac{\partial T}{\partial t} - \lambda \nabla^2 T = s. \quad (1)$$

Here λ is a constant, and s is a source that typically is a known function of time and space.

The heat equation is of fundamental importance in diverse scientific fields. In mathematics, it is the prototypical parabolic partial differential equation. In probability theory, the heat equation is connected with the study of Brownian motion via the Fokker–Planck equation. The diffusion equation, a more general version of the heat equation, arises in connection with the study of chemical diffusion and other related processes. To numerically solve Eq.(1), we re-write the equation in the form of heat flux

$$\frac{\partial T}{\partial t} + \nabla \cdot J = s. \quad (2)$$

Here J is the heat flux defined as

$$J \equiv -\nabla(\lambda T). \quad (3)$$

Here λ is constant for each material, but it may be very different for different materials.

4. Numerical Methods

Considering a numerical grid $\{x_i, y_j, z_k\}$ in three dimensions, we integrate Eq.(2) over one time step $0 < t < \Delta t$ and one numerical cell $x_i < x < x_{i+1}$, $y_j < y < y_{j+1}$, and $z_k < z < z_{k+1}$. Here Δt is the size of time step. The result of the integration may be written as

$$T^n_{i,j,k} = T_{i,j,k} + \frac{\Delta t}{\Delta x}(\bar{J}_{xi,j,k} - \bar{J}_{xi+1,j,k}) + \frac{\Delta t}{\Delta y}(\bar{J}_{yi,j,k} - \bar{J}_{yi,j+1,k}) + \frac{\Delta t}{\Delta z}(\bar{J}_{zi,j,k} - \bar{J}_{zi,j,k+1}) + \bar{s}\Delta t. \quad (4)$$

Here Δx, Δy, and Δz are the widths of the cell in three dimensions. $T_{i,j,k}$ and $T^n_{i,j,k}$ are the space-average of T over the cell at $t = 0$ and $t = \Delta t$, J_{xi}, J_{yj}, and J_{zk} are time-averaged fluxes, $\bar{s}$ is the time- and space-averaged source, and they are defined as

$$T^n_{i,j,k} \equiv \frac{1}{\Delta V}\int_{\Delta V} T(\Delta t, x, y, z)dV, \quad (5)$$

$$\bar{J}_{xi,j,k} \equiv \frac{1}{\Delta t \Delta y \Delta z}\int_0^{\Delta t}\int_{y_j}^{y_{j+1}}\int_{z_k}^{z_{k+1}} J_x(t, x_i, y, z)dydzdt, \quad (6a)$$

$$\bar{J}_{yi,j,k} \equiv \frac{1}{\Delta t \Delta x \Delta z}\int_0^{\Delta t}\int_{x_i}^{x_{i+1}}\int_{z_k}^{z_{k+1}} J_y(t, x, y_j, z)dxdzdt, \quad (6b)$$

$$\bar{J}_{zi,j,k} \equiv \frac{1}{\Delta t \Delta x \Delta y} \int_0^{\Delta t} \int_{x_i}^{x_{i+1}} \int_{y_j}^{y_{j+1}} J_z(t, x, y, z_k) dx dy dt. \qquad (6c)$$

In Eq.(4), the superscript n stands for "new" time $t = \Delta t$. Eq.(4) is exact, and no approximations have been introduced yet. If we know the way to calculate fluxes needed in Eq.(4), we can find the temperature at the new time $t = \Delta t$ from the given initial condition and source function. Therefore, one of the major tasks in the numerical methods is to approximately find the fluxes listed in Eqs.(6a-6c).

If the time-averaged flux in Eqs.(6a-6c) is replaced by a value at the beginning of time step $t = 0$, the approximation results in Euler forward method. If the fluxes are replaced by the values at the end of the time step $t = \Delta t$, the scheme is called Euler backward scheme. Both Euler forward and backward schemes are first order accurate. Euler forward method is an explicit method, and therefore is simple. But the size of time step is limited by a stability condition that is normally much smaller than the required accuracy. Therefore, an explicit scheme may be inefficient for many problems. On the other hand, Euler backward scheme is an implicit method in which the size of time step is not limited by any stability condition, and therefore the time steps may be changed according to the requirement of the problem itself. Numerical errors in Euler backward method undergo quick damping for large time steps that is very useful for steady state problems. But an implicit method normally involves solving a large set of algebraic equations at each time step.

If the time-average fluxes in Eqs.(6a,6b,6c) are placed by their averaged values at $t = 0$ and $t = \Delta t$, the result is the Crank-Nicolson scheme, which is second order in time. But the numerical errors in Crank-Nicolson method do not damp out for large time steps. The reason for the absence of the damping is that when the size of time step is very large, the solution should be independent on the initial condition of a problem, and the solution is determined only by boundary conditions of the problem. But in Crank-Nicolson method, the calculation for fluxes is based on the values at $t = \Delta t$, as well as the initial values. It is this understanding that lead us to develop a new numerical method that will give exact steady states.

There are typically two approaches to solve the large set of algebraic equations, i.e., direct approach and iterative approach. Direct approach is to find the exact solution of the set of algebraic equations. In general, the direct approach is computationally expensive. For any iterative approach [4], a significant question is whether an iterative process will actually be successful and will lead to the solution of the set of algebraic equations. An important question related to an iterative approach is the rate of convergence.

As stated before, numerical errors in Crank-Nicolson method do not damped out when the size of time step is large, although the method is second order accurate. To introduce quick damping for numerical

errors within the second order accurate, we introduce an additional time step level $t = \Delta t/2$. Within the second order accuracy, we approximately evaluate the time-averaged flux at $t = \Delta t/2$, and Eq.(4) becomes

$$T^n_{i,j,k} = T_{i,j,k} + \frac{\Delta t}{\Delta x}(J^h_{xi,j,k} - J^h_{xi+1,j,k}) + \frac{\Delta t}{\Delta y}(J^h_{yi,j,k} - J^h_{yi,j+1,k}) + \frac{\Delta t}{\Delta z}(J^h_{zi,j,k} - J^h_{zi,j,k+1}) + s^h \Delta t. \quad (7)$$

Here the superscript h stands for the evaluation at a half time step, $t = \Delta t/2$, and the fluxes at the half-time step are defined

$$J^h_{xi,j,k} \equiv \frac{1}{\Delta t \Delta y \Delta z}\int_0^{\Delta t}\int_{y_j}^{y_{j+1}}\int_{z_k}^{z_{k+1}} J_x(\Delta t/2, x_i, y, z)\,dydzdt, \quad (8a)$$

$$J^h_{yi,j,k} \equiv \frac{1}{\Delta t \Delta x \Delta z}\int_0^{\Delta t}\int_{x_i}^{x_{i+1}}\int_{z_k}^{z_{k+1}} J_y(\Delta t/2, x, y_j, z)\,dxdzdt, \quad (8b)$$

$$J^h_{zi,j,k} \equiv \frac{1}{\Delta t \Delta x \Delta y}\int_0^{\Delta t}\int_{x_i}^{x_{i+1}}\int_{y_j}^{y_{j+1}} J_z(\Delta t/2, x, y, z_k)\,dxdydt. \quad (8c)$$

As shown in Eq.(4), the equations to determine the temperature at the half time step may be similarly obtained,

$$T^h_{i,j,k} = T_{i,j,k} + \frac{\Delta t}{2\Delta x}(\bar{J}^h_{xi,j,k} - \bar{J}^h_{xi+1,j,k}) + \frac{\Delta t}{2\Delta y}(\bar{J}^h_{yi,j,k} - \bar{J}^h_{yi,j+1,k}) + \frac{\Delta t}{2\Delta z}(\bar{J}^h_{zi,j,k} - \bar{J}^h_{zi,j,k+1}) + \frac{1}{2}\bar{s}^h \Delta t. \quad (9)$$

Here $T^h_{i,j,k}$ and fluxes are defined as

$$T^h_{i,j,k} \equiv \frac{1}{\Delta V}\int_{\Delta V} T(\Delta t/2, x, y, z)\,dV, \quad (10)$$

$$\bar{J}^h_{xi,j,k} \equiv \frac{2}{\Delta t \Delta y \Delta z}\int_0^{\Delta t/2}\int_{y_j}^{y_{j+1}}\int_{z_k}^{z_{k+1}} J_x(t, x_i, y, z)\,dydzdt, \quad (11a)$$

$$\bar{J}^h_{yi,j,k} \equiv \frac{2}{\Delta t \Delta x \Delta z}\int_0^{\Delta t/2}\int_{x_i}^{x_{i+1}}\int_{z_k}^{z_{k+1}} J_y(t, x, y_j, z)\,dxdzdt, \quad (11b)$$

$$\bar{J}^h_{zi,j,k} \equiv \frac{2}{\Delta t \Delta x \Delta y}\int_0^{\Delta t/2}\int_{x_i}^{x_{i+1}}\int_{y_j}^{y_{j+1}} J_z(t, x, y, z_k)\,dxdydt. \quad (11c)$$

The time-averaged flux involved in Eq.(9) may be approximately calculated through an interpolation in time. But, as stated before, an approximate calculation for the time-averaged flux must not, even partially, depend on the initial information when a time step is very large. Therefore, our interpolation for the time-

averaged flux is uniquely determined by values at $t = \Delta t/2$ and $t = \Delta t$. Thus, the time-averaged fluxes in Eq.(9) are approximately obtained through the following approximation:

$$\bar{J}^h_{xi,j,k} \approx \frac{3}{2} J^h_{xi,j,k} - \frac{1}{2} J^n_{xi,j,k}, \quad (12a)$$

$$\bar{J}^h_{yi,j,k} \approx \frac{3}{2} J^h_{yi,j,k} - \frac{1}{2} J^n_{yi,j,k}, \quad (12b)$$

$$\bar{J}^h_{zi,j,k} \approx \frac{3}{2} J^h_{zi,j,k} - \frac{1}{2} J^n_{zi,j,k}. \quad (12c)$$

Here $J^n_{xi,j,k}$, $J^n_{yi,j,k}$, and $J^n_{zi,j,k}$ have the same definitions as Eqs.(8a-8c) except for that $J^n_{xi,j,k}$, $J^n_{yi,j,k}$, and $J^n_{zi,j,k}$ are evaluated at $t = \Delta t$ instead of $t = \Delta t/2$.

To give specific forms of the flux at cell interfaces and material interfaces, we write the numerical fluxes for the heat flux defined in Eq.(3). If the material in the cell (i,j) is the same as those in its neighboring cells, the central difference may be used to approximately calculate the flux at cell interfaces, for example,

$$J^h_{xi,j,k} \approx -\frac{\lambda}{\Delta x}(T^h_{i,j,k} - T^h_{i-1,j,k}). \quad (13)$$

But, since the material in one cell may be thermally very different from the materials in the neighboring cells, the derivative of temperature is no longer continuous across material interfaces. Therefore, Eq.(13) is no longer true across material interfaces across material interface.

To find a correct formula valid for material interfaces, assume T^h_* the temperature at the interface between cells (i-1,j,k) and (i,j,k). Since material is the same at each side of the material interface, the values of the fluxes calculated from each side of the interface is approximately

$$J^{h-}_{xi,j,k} \approx -\frac{2}{\Delta x}(\lambda_{i-1,j,k} T_* - \lambda_{i-1,j,k} T^h_{i-1,j,k}). \quad (14)$$

$$J^{h+}_{xi,j,k} \approx -\frac{2}{\Delta x}(\lambda_{i,j,k} T^h_{i,j,k} - \lambda_{i,j,k} T_*). \quad (15)$$

Since heat is conserved across material interface, it must be true that $J^{h-}_{xi,j,k} = J^{h+}_{xi,j,k}$. Through this equation we may find T_*,

$$T_* = \frac{\lambda_{i-1,j,k} T^h_{i-1,j,k} + \lambda_{i,j,k} T^h_{i,j,k}}{\lambda_{i-1,j,k} + \lambda_{i,j,k}}.$$

Putting this back into Eq.(14) or Eq.(15), we have the correct flux across the interface

$$J^h_{xi} \approx -\frac{\lambda_{xi,j,k}}{\Delta x}[T^h_{i,j,k} - T^h_{i-1,j,k}]. \quad (16)$$

Here $\lambda_{xi,j,k}$ is defined as

$$\lambda_{xi,j,k} \equiv \frac{2\lambda_{i-1,j,k}\lambda_{i,j,k}}{\lambda_{i-1,j,k}+\lambda_{i,j,k}}. \quad (17)$$

Equation (25) is the extension of Eq.(22) that is valid for different materials across the interface. Similarly, we may find other fluxes,

$$J^h_{yj} \approx -\frac{\lambda_{yi,j,k}}{\Delta y}[T^h_{i,j,k} - T^h_{i,j-1,k}], \quad (18)$$

$$J^h_{zk} \approx -\frac{\lambda_{zi,j,k}}{\Delta z}[T^h_{i,j,k} - T^h_{i,j,k-1}]. \quad (19)$$

Here $\lambda_{yi,j,k}$ and $\lambda_{zi,j,k}$ are defined as the following

$$\lambda_{yi,j,k} \equiv \frac{\lambda_{i,j-1,k}\lambda_{i,j,k}}{\lambda_{i,j-1,k}+\lambda_{i,j,k}}, \quad (20)$$

$$\lambda_{zi,j,k} \equiv \frac{\lambda_{i,j,k-1}\lambda_{i,j,k}}{\lambda_{i,j,k-1}+\lambda_{i,j,k}}. \quad (21)$$

Applying the fluxes in Eqs.(16,18,19) to Eqs.(12a-12c), and applying the result to Eqs.(7,9), we arrive at the following set of equations for $T^n_{i,j,k}$ and $T^h_{i,j,k}$,

$$T^n_{i,j,k} = T_{i,j,k} + s^h\Delta t + D^h_{i,j,k}, \quad (22a)$$

$$T^h_{i,j,k} = T_{i,j,k} + (\frac{3}{4}s^h - \frac{1}{4}s^n)\Delta t + \frac{3}{4}D^h_{i,j,k} - \frac{1}{4}D^n_{i,j,k}. \quad (22b)$$

Here $D^h_{i,j,k}$ is defined as

$$\begin{aligned} D^h_{i,j,k} \equiv{}& \frac{\Delta t}{(\Delta x)^2}[\lambda_{xi,j,k}T^h_{i-1,j,k} + \lambda_{xi+1,j,k}T^h_{i+1,j,k} - (\lambda_{xi,j,k}+\lambda_{xi+1,j,k})T^h_{i,j,k}] \\ &+ \frac{\Delta t}{(\Delta y)^2}[\lambda_{yi,j,k}T^h_{i,j-1,k} + \lambda_{yi,j+1,k}T^h_{i,j+1,k} - (\lambda_{yi,j,k}+\lambda_{yi,j+1,k})T^h_{i,j,k}] \\ &+ \frac{\Delta t}{(\Delta z)^2}[\lambda_{zi,j,k}T^h_{i,j,k-1} + \lambda_{zi,j,k+1}T^h_{i,j,k+1} - (\lambda_{zi,j,k}+\lambda_{zi,j,k+1})T^h_{i,j,k}]. \quad (23) \end{aligned}$$

$D^n_{i,j,k}$ has the same form as $D^h_{i,j,k}$ except for the superscript h to be replaced by n.

Numerical scheme, Eqs.(22a, 22b) together with Eq.(23) and similar $D^n_{i,j,k}$, is second order accurate in space and time, and it is valid for systems with any number of materials no matter how different the thermal properties of the materials are. Furthermore, unlike Crank-Nicolson method, this scheme damps out numerical errors when the size of time step is large. Therefore the scheme we just derived is appropriate for both transient and steady states.

Let us discuss three special cases of Eq.(22). The first one is for single material. In this case, λ_x, λ_y, and λ_z all become λ, and $D^h_{i,j,k}$ and $D^n_{i,j,k}$become

$$D^h_{i,j,k} \equiv \frac{\lambda\Delta t}{(\Delta x)^2}[T^h_{i-1,j,k} - T^h_{i,j,k} + T^h_{i+1,j,k} - T^h_{i,j,k}] + \frac{\lambda\Delta t}{(\Delta y)^2}[T^h_{i,j-1,k} - T^h_{i,j,k} + T^h_{i,j+1,k} - T^h_{i,j,k}] + \frac{\lambda\Delta t}{(\Delta z)^2}[T^h_{i,j,k-1} - T^h_{i,j,k} + T^h_{i,j,k+1} - T^h_{i,j,k}], \quad (24)$$

$$D^n_{i,j,k} \equiv \frac{\lambda\Delta t}{(\Delta x)^2}[T^n_{i-1,j,k} - T^n_{i,j,k} + T^n_{i+1,j,k} - T^n_{i,j,k}] + \frac{\lambda\Delta t}{(\Delta z)^2}[T^h_{i,j,k-1} - T^h_{i,j,k} + T^h_{i,j,k+1} - T^h_{i,j,k}],$$
$$+ \frac{\lambda\Delta t}{(\Delta z)^2}[T^n_{i,j,k-1} - T^n_{i,j,k} + T^n_{i,j,k+1} - T^n_{i,j,k}]. \quad (25)$$

Eqs.(22) together with Eqs.(24,25) are appropriate for heat conduction of single material, second order accurate in both space and time for transient problem, and give steady states when the size of time step is very large.

The second special case is for the steady state, for which Eq.(22) becomes

$$\frac{1}{(\Delta x)^2}[\lambda_{xi,j,k}T^h_{i-1,j,k} + \lambda_{xi+1,j,k}T^h_{i+1,j,k} - (\lambda_{xi,j,k} + \lambda_{xi+1,j,k})T^h_{i,j,k}]$$
$$+ \frac{1}{(\Delta y)^2}[\lambda_{yi,j,k}T^h_{i,j-1,k} + \lambda_{yi,j+1,k}T^h_{i,j+1,k} - (\lambda_{yi,j,k} + \lambda_{yi,j+1,k})T^h_{i,j,k}]$$
$$+ \frac{1}{(\Delta z)^2}[\lambda_{zi,j,k}T^h_{i,j,k-1} + \lambda_{zi,j,k+1}T^h_{i,j,k+1} - (\lambda_{zi,j,k} + \lambda_{zi,j,k+1})T^h_{i,j,k}] = 0. \quad (26)$$

This equation will give the steady state for system with any number of materials.

The third special case is for the steady state with a single material. For which the scheme is simplified to

$$2[\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2}]T^n_{i,j,k} = \frac{1}{(\Delta x)^2}[T^n_{i-1,j,k} + T^n_{i+1,j,k}] + \frac{1}{(\Delta y)^2}[T^n_{i,j-1,k} + T^n_{i,j+1,k}] + \frac{1}{(\Delta z)^2}[T^n_{i,j,k-1} + T^n_{i,j,k+1}]. \quad (27)$$

5. Iterative Solvers

Equations (22a,22b) may be iteratively solved. It seems that we may use the evaluation of the right hand sides of Eqs.(22a,22b) through an initial guess of T at $t = \Delta t$ and $t = \Delta t/2$ to find an improved the temperatures $T^n_{i,j,k}$ and $T^h_{i,j,k}$. Unfortunately, this iterative approach doesn't converge when $\lambda\Delta t/(\Delta x)^2$ is larger than 1. Through each iteration, numerical errors in $T^n_{i,j,k}$ and $T^h_{i,j,k}$ are increased by a factor larger than 1 when $\lambda\Delta t/(\Delta x)^2$ is larger than unity.

To find a successful iterative approach, we move the terms with $T^h_{i,j,k}$ and $T^h_{i,j,k}$ to the left side and write Eqs.(22a,22b) in the form

$$T^n_{i,j,k} + \xi_{i,j,k}T^h_{i,j,k} = T_{i,j,k} + s^h_{i,j,k}\Delta t + Q^h_{i,j,k}, \quad (28a)$$

$$-\frac{1}{4}\xi_{i,j,k}T^n_{i,j,k} + (1 + \frac{3}{4}\xi_{i,j,k})T^h_{i,j,k} = T_{i,j,k} + (\frac{3}{4}s^h_{i,j,k} - \frac{1}{4}s^n_{i,j,k})\Delta t + \frac{3}{4}Q^h_{i,j,k} - \frac{1}{4}Q^n_{i,j,k}. \quad (28b)$$

Here $\xi_{i,j,k}$, $Q^h_{i,j,k}$, and $Q^n_{i,j,k}$ are defines as

$$\xi_{i,j,k} \equiv \frac{\Delta t}{(\Delta x)^2}(\lambda_{xi,j,k} + \lambda_{xi+1,j,k}) + \frac{\Delta t}{(\Delta y)^2}(\lambda_{yi,j,k} + \lambda_{yi,j+1,k}) + \frac{\Delta t}{(\Delta z)^2}(\lambda_{zi,j,k} + \lambda_{zi,j,k+1}), \qquad (29)$$

$$Q^h_{i,j,k} \equiv \frac{\Delta t}{(\Delta x)^2}(\lambda_{xi,j,k}T^h_{i-1,j,k} + \lambda_{xi+1,j,k}T^h_{i+1,j,k}) + \frac{\Delta t}{(\Delta y)^2}(\lambda_{yi,j,k}T^h_{i,j-1,k} + \lambda_{yi,j+1,k}T^h_{i,j+1,k}) + \frac{\Delta t}{(\Delta z)^2}(\lambda_{zi,j,k}T^h_{i,j,k-1} + \lambda_{zi,j,k+1}T^h_{i,j,k+1}), \qquad (30a)$$

$$Q^n_{i,j,k} \equiv \frac{\Delta t}{(\Delta x)^2}(\lambda_{xi,j,k}T^n_{i-1,j,k} + \lambda_{xi+1,j,k}T^n_{i+1,j,k}) + \frac{\Delta t}{(\Delta y)^2}(\lambda_{yi,j,k}T^n_{i,j-1,k} + \lambda_{yi,j+1,k}T^n_{i,j+1,k}) + \frac{\Delta t}{(\Delta z)^2}(\lambda_{zi,j,k}T^n_{i,j,k-1} + \lambda_{zi,j,k+1}T^n_{i,j,k+1}). \qquad (30b)$$

We solve Eqs.(28a, 28b) for $T^n_{i,j,k}$ and $T^h_{i,j,k}$,

$$T^n_{i,j,k} = \frac{1}{A}[(1 - \frac{1}{4}\xi_{i,j,k})T_{i,j,k} + (s^h_{i,j,k} + \frac{1}{4}\xi_{i,j,k}s^n_{i,j,k})\Delta t + Q^h_{i,j,k} + \frac{1}{4}\xi_{i,j,k}Q^n_{i,j,k}], \qquad (31a)$$

$$T^h_{i,j,k} = \frac{1}{A}[(1 + \frac{1}{4}\xi_{i,j,k})T_{i,j,k} + \frac{1}{4}(3 + \xi_{i,j,k})s^h_{i,j,k}\Delta t - \frac{1}{4}s^n_{i,j,k}\Delta t + \frac{1}{4}(3 + \xi_{i,j,k})Q^h_{i,j,k} - \frac{1}{4}Q^n_{i,j,k}]. \qquad (31b)$$

Here A is defines as

$$A \equiv 1 + \frac{1}{4}\xi_{i,j,k}(3 + \xi_{i,j,k}).$$

Our iterative procedure is as follows. We initially guess the temperature on all the cells at $t = \Delta t/2$ and $t = \Delta t$. Then we evaluate the right hand side of Eqs.(31a,31b). The improved the temperatures at $t = \Delta t/2$ and $t = \Delta t$ are thus obtained. If the improved temperatures do not satisfy the accuracy requirement, we may consider the improved solution as an initial guess to continue the iteration. This primitive iterative approach is called Gauss-Seidel method. Numerical experiments show that this iterative procedure converges. An improvement of the convergence rate to Gauss-Seidel method could be made through red-black method. In the red-black method, cells are divided into two staggered sets, red and black. The two sets are alternatively updated through iterations.

6. Multigrid Method

From numerical experiments for iteratively solve Eq.(1) for a constant thermal conductivity, it is well know that numerical errors with high frequencies are efficiently killed in the first few iterations, but errors with low frequencies remain even after many iterations. These phenomena indicate that we may use a coarse grid to kill low frequency errors, ie, we may use the multigrid method to speed up the convergence [5-8,11].

A typical algorithm of the multigrid method starts from a finer grid. After a few iterations, the grid cells in each direction are halved, resulting in a coarser grid. Another set of a few iterations is implemented in the coarser grid with the initial guess obtained from the solution on the finer grid. Therefore, numerical errors with low frequencies are significantly killed in the coarser grid, and the errors with high frequencies are significantly killed in the finer grid.

Since the coarse grid is more efficient for a smoother solution, we may work on the residue of the solution instead of the solution itself on the coarser grid. Suppose that $c^n_{i,j,k}$ and $c^h_{i,j,k}$ are errors or corrections, and that $r^n_{i,j,k}$ and $r^h_{i,j,k}$ are residues of Eqs.(28a,b),

$$c^n_{i,j,k} \equiv T^n_{i,j,k} - e^n_{i,j,k}, \quad (44)$$

$$c^h_{i,j,k} \equiv T^h_{i,j,k} - e^h_{i,j,k}, \quad (32b)$$

$$r^n_{i,j,k} \equiv T^n_{i,j,k} + \xi_{i,j,k} T^h_{i,j,k} - [T_{i,j,k} + s^h_{i,j,k}\Delta t + Q^h_{i,j,k}], \quad (33a)$$

$$r^h_{i,j,k} \equiv -\frac{1}{4}\xi_{i,j,k} T^n_{i,j,k} + (1+\frac{3}{4}\xi_{i,j,k}) T^h_{i,j,k} - [T_{i,j,k} + (\frac{3}{4}s^h_{i,j,k} - \frac{1}{4}s^n_{i,j,k})\Delta t + \frac{3}{4}Q^h_{i,j,k} - \frac{1}{4}Q^n_{i,j,k}]. \quad (33b)$$

Here $e^n_{i,j,k}$ and $e^h_{i,j,k}$ are the exact solution of Eqs.(28a,b). Substituting $e^n_{i,j,k} = T^n_{i,j,k} - c^n_{i,j,k}$ and $e^h_{i,j,k} = T^h_{i,j,k} - c^h_{i,j,k}$ into Eqs.(28a,b), we have

$$c^n_{i,j,k} + \xi_{i,j,k} c^h_{i,j,k} = Q^h_{i,j,k}(c), \quad (34a)$$

$$-\frac{1}{4}\xi_{i,j,k} c^n_{i,j,k} + (1+\frac{3}{4}\xi_{i,j,k}) c^h_{i,j,k} = \frac{3}{4}Q^h_{i,j,k}(c) - \frac{1}{4}Q^n_{i,j,k}(c). \quad (34b)$$

Here $Q^n_{i,j,k}(c)$ and $Q^h_{i,j,k}(c)$ are $Q^n_{i,j,k}$ and $Q^h_{i,j,k}$ defined in Eqs.(30a,b) but evaluated at $c^n_{i,j,k}$ and $c^h_{i,j,k}$ instead of $T^n_{i,j,k}$ and $T^h_{i,j,k}$. Eqs.(34a,b) may be iteratively solved in the exactly same way as we do with Eqs.(28a,b). A few iterations of Eqs.(34a,b) on the coarser grid will give very accurate solutions for $c^n_{i,j,k}$ and $c^h_{i,j,k}$, since $c^n_{i,j,k}$ and $c^h_{i,j,k}$ are smooth and vanishing $c^n_{i,j,k}$ and $c^h_{i,j,k}$ are reasonable initial guess. After a few iterations for $c^n_{i,j,k}$ and $c^h_{i,j,k}$ on the coarser grid, we may correct the solution on the finer grid by subtracting $c^n_{i,j,k}$ and $c^h_{i,j,k}$ from $T^n_{i,j,k}$ and $T^h_{i,j,k}$,

$$T^n_{i,j,k} = T^n_{i,j,k} - c^n_{i,j,k},$$

$$T^h_{i,j,k} = T^h_{i,j,k} - c^h_{i,j,k}.$$

This procedure is often called “coarse grid correction”. Typically, the solution after the coarser grid correction contains large errors with high frequencies for which a few iterations on the finer grid are needed.

In the multigrid method described above, we have to map $c^n_{i,j,k}$ and $c^h_{i,j,k}$, or $T^n_{i,j,k}$ and $T^h_{i,j,k}$ between the finer and coarser grids. For the mapping from the finer grid to coarser one, by definition, we have to only add cell-averaged $T^n_{i,j,k}$ and $T^h_{i,j,k}$ defined on a set of fine cells together in order to find cell-averaged $T^n_{i,j,k}$ and $T^h_{i,j,k}$ on the coarser grid. For the mapping from the coarser grid to finer grid, the simplest interpolation is to assume no internal structures for $c^n_{i,j,k}$ and $c^h_{i,j,k}$ within cells, i.,e., they are piecewise constants on the coarser grid. We may also assume piecewise linear or parabolic internal structure within the cells on the coarser grid. The mapping based on internal structures will result in more accurate initial guess on the finer grid than the mapping with piecewise constants. But, the difference in the initial guess on the finer grid between the piecewise constant interpolation and piecewise linear (or parabolic) interpolation are dominated by high frequencies. The difference will be immediately killed in the first few of iterations on

the finer grid. Therefore, there will be no difference in the convergence rate between the different interpolations for the mapping.

7. Numerical Examples

In this section we will provide numerical examples to show the correctness of the schemes for single and multi-materials and properties of scheme, including rapid convergence of the multigrid method and effectiveness for steady states for large time steps. When time steps are small, the scheme is second order accurate in both space and time. When a time step is very large, the solution of the scheme will give the correct steady state, which is second order accurate in space. The examples to be given in this section include one-, two- and three-dimensional situations. The purpose of these examples is to demonstrate the desired features of the scheme.

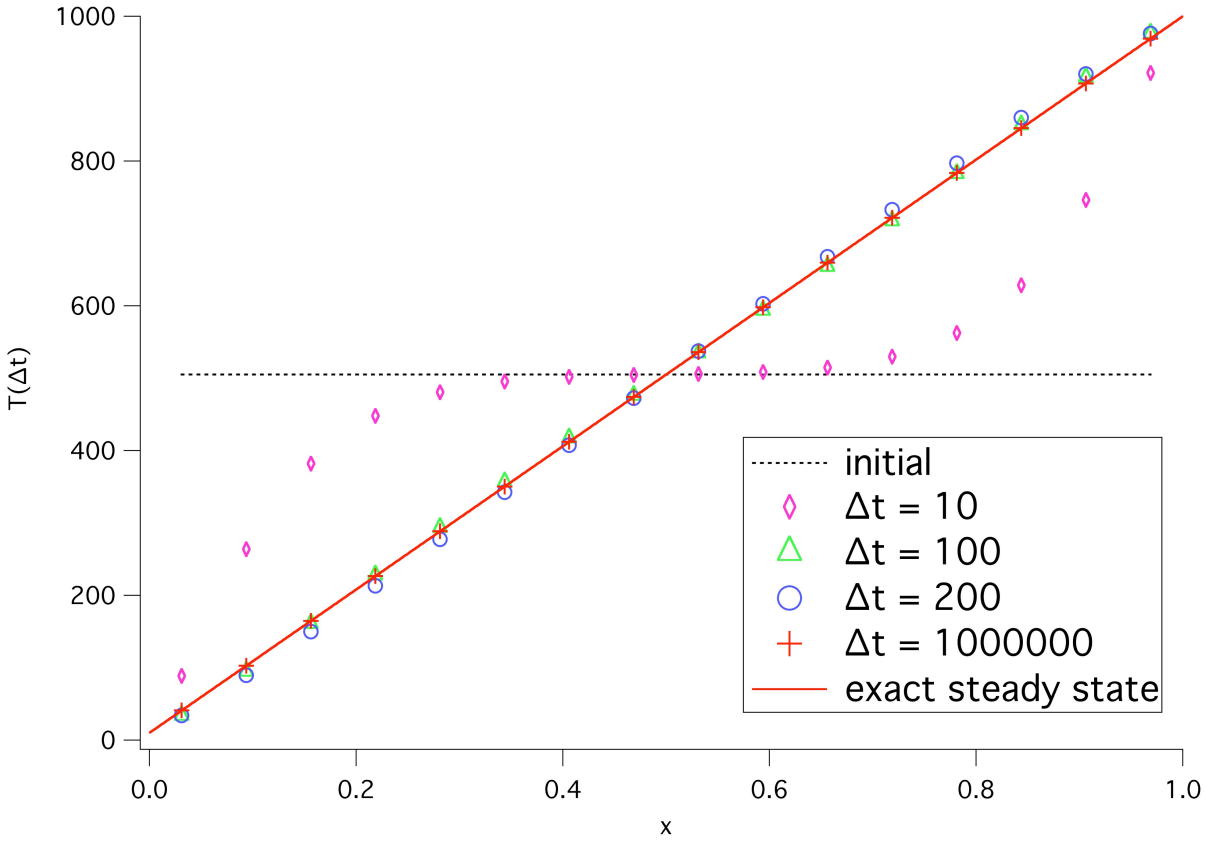

Figure 1. Nmerical solutions after one time step. The dashed line is the initial condition, and the solid line is the exact steady state. Sixteen grid cells are used in these one-dimensional simulations.

The first example is to show how the solutions of the scheme change with different time steps. Figure 1 shows four solutions obtained from one-dimensional simulations with sixteen grid cells and fixed temperatures at x = 0 and x = 1.0 after one time step. The dashed line is the initial distribution of

temperature. The diamonds are the solution after one time step Δt = 10, and they are approximate to the exact solution at t = 10. The triangles and circles in the figure are the solutions after one time step with Δt = 100 and 200, and the crosses are the solution after a very large time step. The solid red line is the exact solution of the steady state of the problem. As expected, the solution after a very large time step is the correct steady state.

The second problem is to show the correctness when two material are involved. Figure 2 shows a solution after a very large time step of a one-dimensional simulation with 32 grid cells, fixed temperatures at boundaries, and two materials, one with thermal conductivity 0.001 (x < 0.5) and the other with thermal conductivity 0.01 (x > 0.5). The dashed time in the figure is the initial temperature and the solid line is the exact steady state of the problem. It should be pointed out that dT/dx is discontinuous at x = 0.5, but heat flux is continuous at the point. We have used this feature in the design of our numerical scheme. It is this feature that makes our scheme very accuracy for problems involving multi-materials.

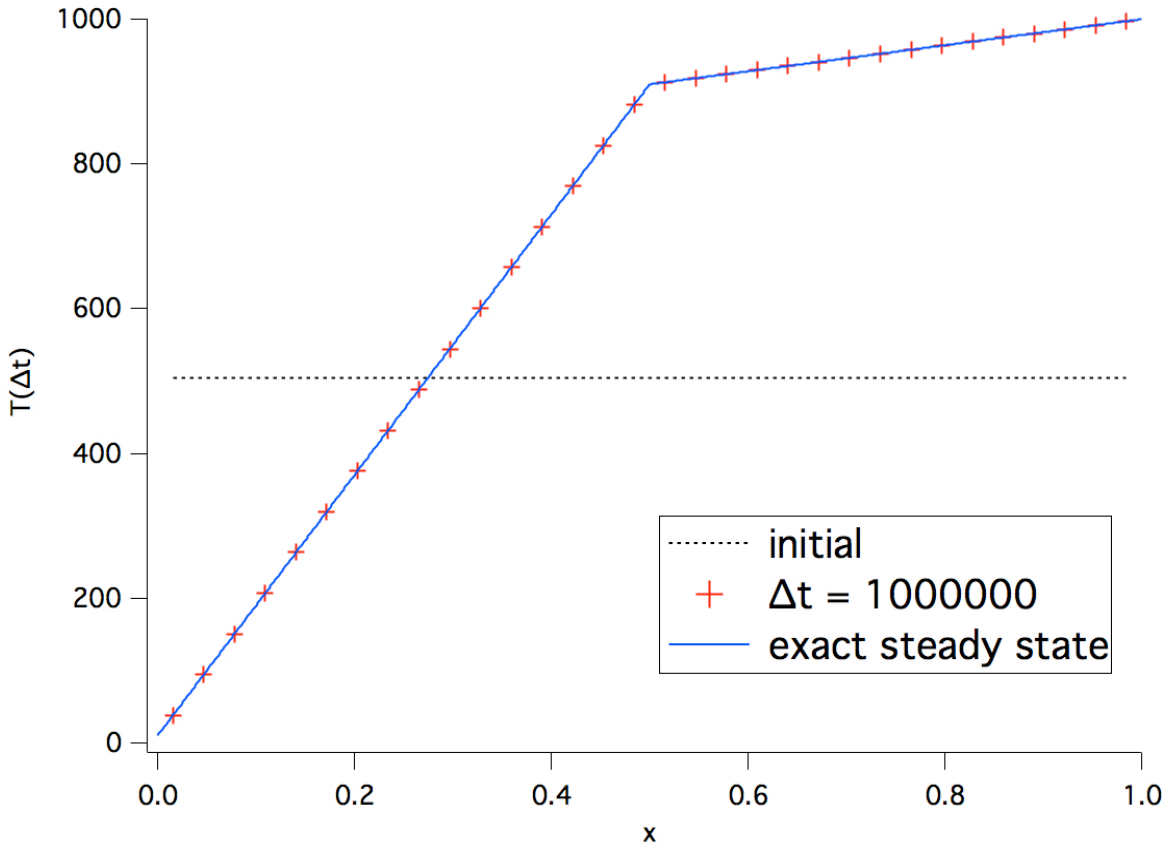

Figure 2. The solution after a very large time step for one-dimensional problem with two materials. Thirty two grid cells are used. The dashed line is the initial condition, and the solid time is the exact steady state.

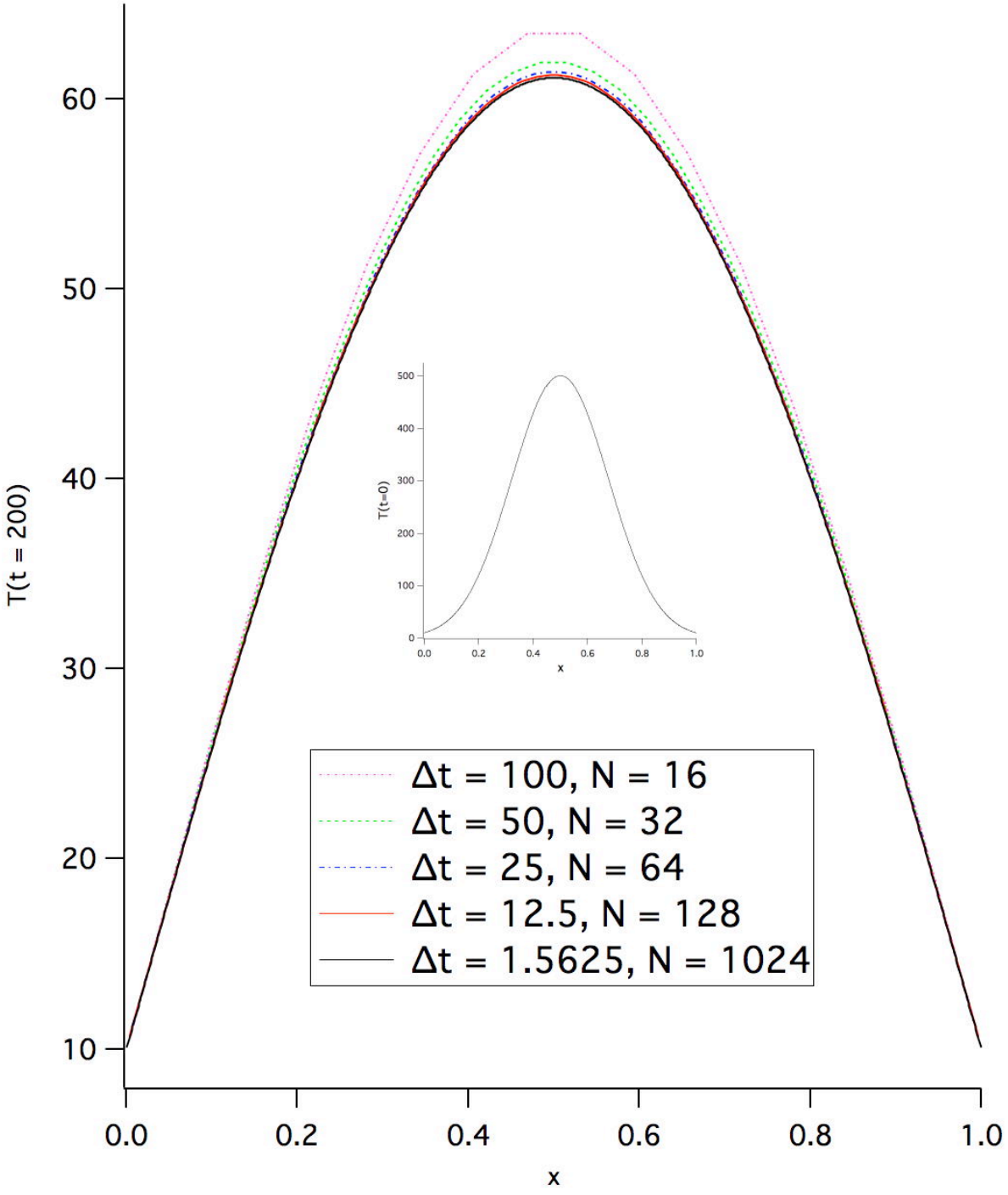

Figure 3. Five different solutions at t = 200 when five different grids and sizes of time step are used. The picture-in-picture is the initial profile.

The third example is to examine the accuracy of the scheme. Figure 3 shows five solutions at t = 200 of one-dimensional problem obtained from five simulations with different numbers of grid cells and time steps. The figure shows that when the width of cell and size of time step are reduced, the solutions

nonlinearly approach to the exact solution if we consider the solid black line is a good approximation to the exact solution.

One-dimensional examples are easy to show the correctness of solutions, and this is the reason we choose one-dimensional problems as examples. The next two examples are to show the effectiveness of multigrid method in two- and three-dimensional cases. Figure 4 shows the convergence rates of the red-black and multigrid method for one time step $\lambda\Delta t/(\Delta x)^2$ = 1638.4. Here Δx is the width of a cell. The vertical axis is the maximum of residues $r_{i,j}^n$ and $r_{i,j}^h$ among all the cells after each iteration. The actual values of the end for the multigrid method are (24, 2.9e-07). We should point out that each iteration of the multigrid method contains a number of iteration on coarse grids. Therefore, one iteration of the multigrid method is more expensive than one iteration in the red-black method in term of CPU time. To show the effectiveness of the multigrid method in terms of CPU time, a comparison between the two methods is given. The actual values at the end of multigrid method in Fig.5 are (0.09, 2.9e-07).

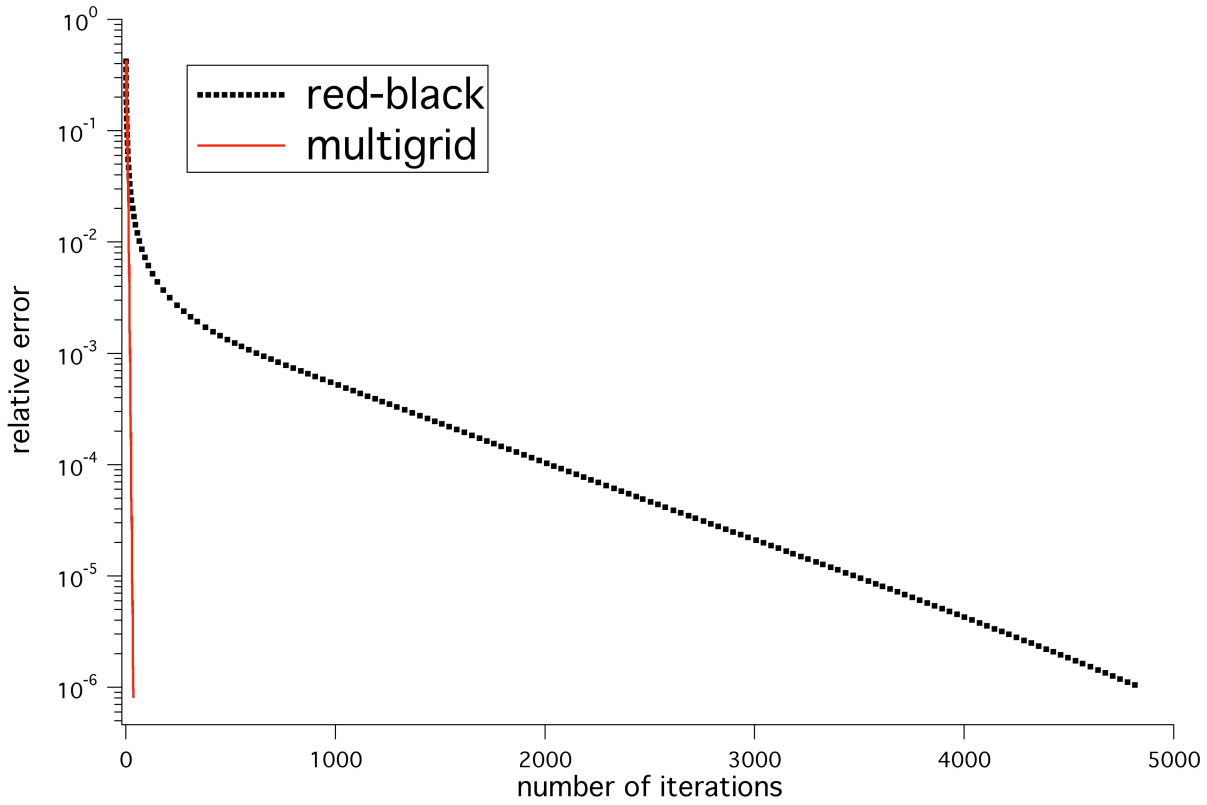

Figure 4. The converence of two iterative approaches, red-black and multigrid method, in a two-dimensional simulations. $\lambda\Delta t/(\Delta x)^2$ = 1638.4.The actual number of iteration for the multgrid method is 24 at the end of the red line.

One-dimensional examples are easy to show the correctness of solutions, and this is the reason we choose one-dimensional problems as examples. The next two examples are to show the effectiveness of

multigrid method in two- and three-dimensional cases. Figure 4 shows the convergence rates of the red-black and multigrid method for one time step $\lambda\Delta t/(\Delta x)^2 = 1638.4$. Here Δx is the width of a cell. The vertical axis is the maximum of residues $r_{i,j}^n$ and $r_{i,j}^h$ among all the cells after each iteration. The actual values of the end for the multigrid method are (24, 2.9e-07). We should point out that each iteration of multigrid method contains many iterations on coarser grids. Therefore, one iteration of multigrid method is more expensive than one iteration in the red-black method in term of CPU time. To show the effectiveness of multigrid method in terms of CPU time, a comparison between the two methods is given in Fig.5. The actual values at the end of multigrid method are (0.09, 2.9e-07).

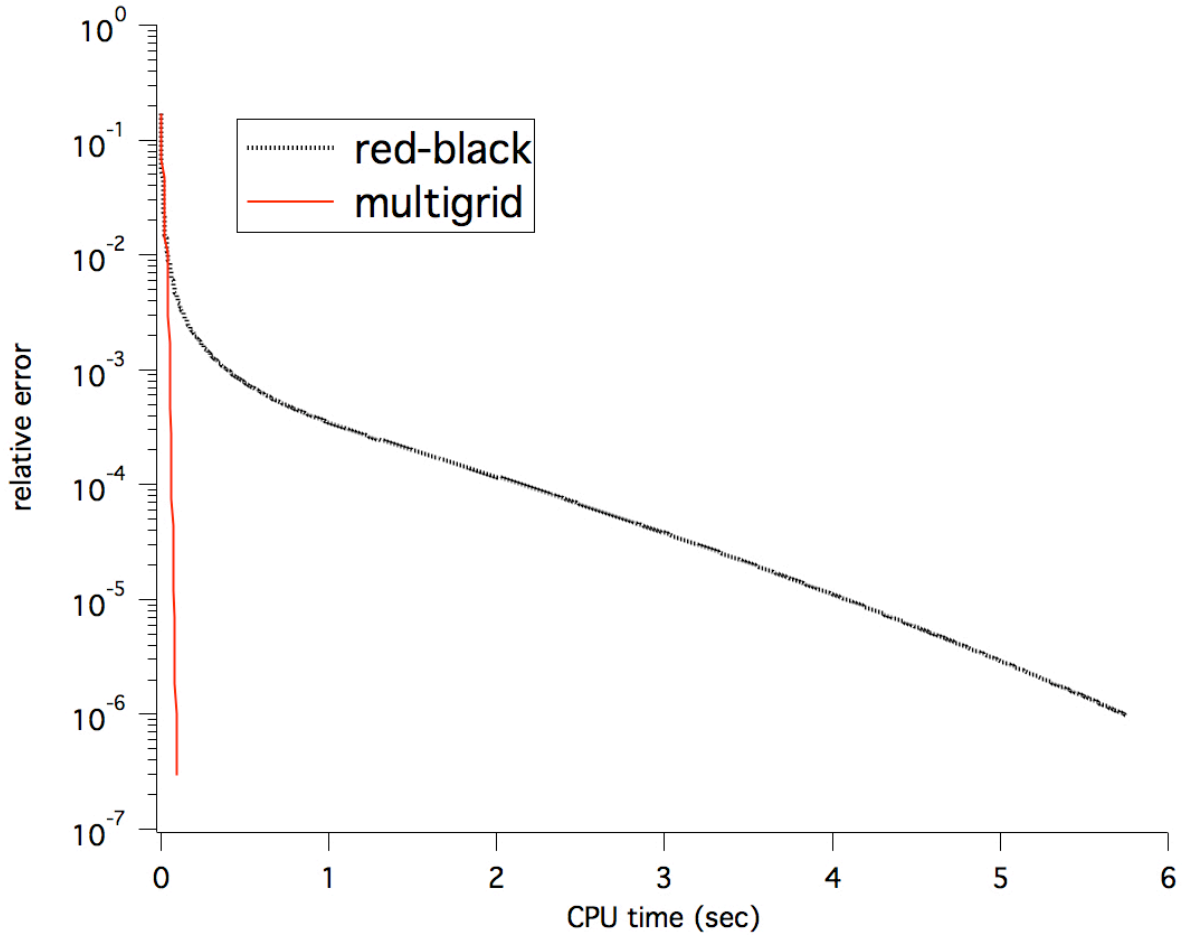

Figure 5. Relative errors vs the CPU time in two iterative approaches, red-black and multigrid, in a two-dimensional simulation. $\lambda\Delta t/(\Delta x)^2 = 1638.4$. The actual CPU time for the multigrid method at the end of the red line is 0.09 sec.

The next example is for the convergence rates for a three-dimensional case with $\lambda\Delta t/(\Delta x)^2 = 1638.4$, which is shown in Figs.6 and 7 in terms of the number of iterations and CPU time. The number of iterations and CPU time at the ends for the multigrid method in Figs.5 and 6 are 36 and 35.25 sec.

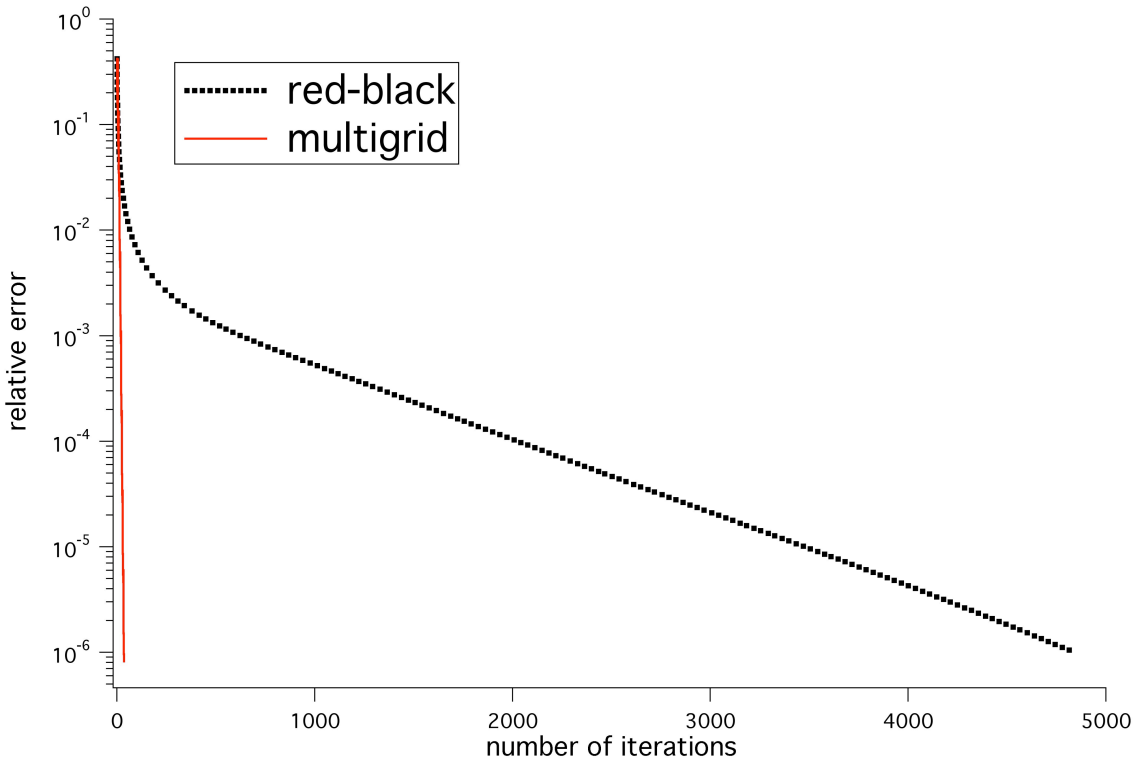

Figure 6. The converence of two iterative approaches, red-black and multigrid method, in a three-dimensional simulations. $\lambda\Delta t/(\Delta x)^2 =$ 1638.4. The actual number of iteration for the multgrid method is 36 at the end of the red line.

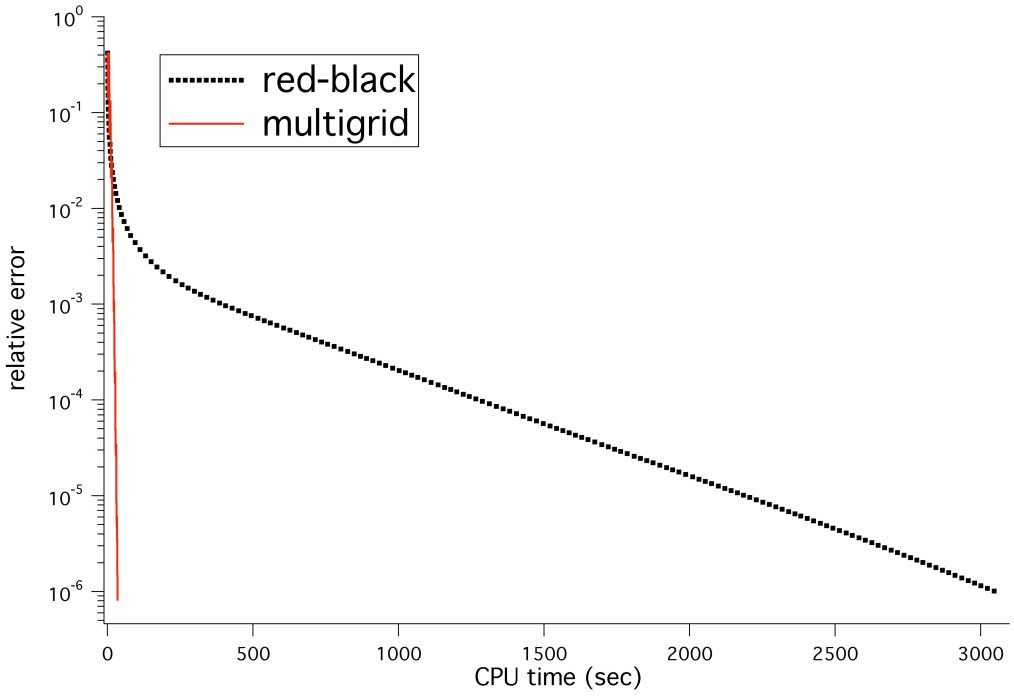

Figure 7. Relative errors vs the CPU time in two iterative approaches, red-black and multigrid, in a three-dimensional simulation. $\lambda\Delta t/(\Delta x)^2 =$ 1638.4. The actual CPU time for the multigrid method at the end of the red line is 35 sec.

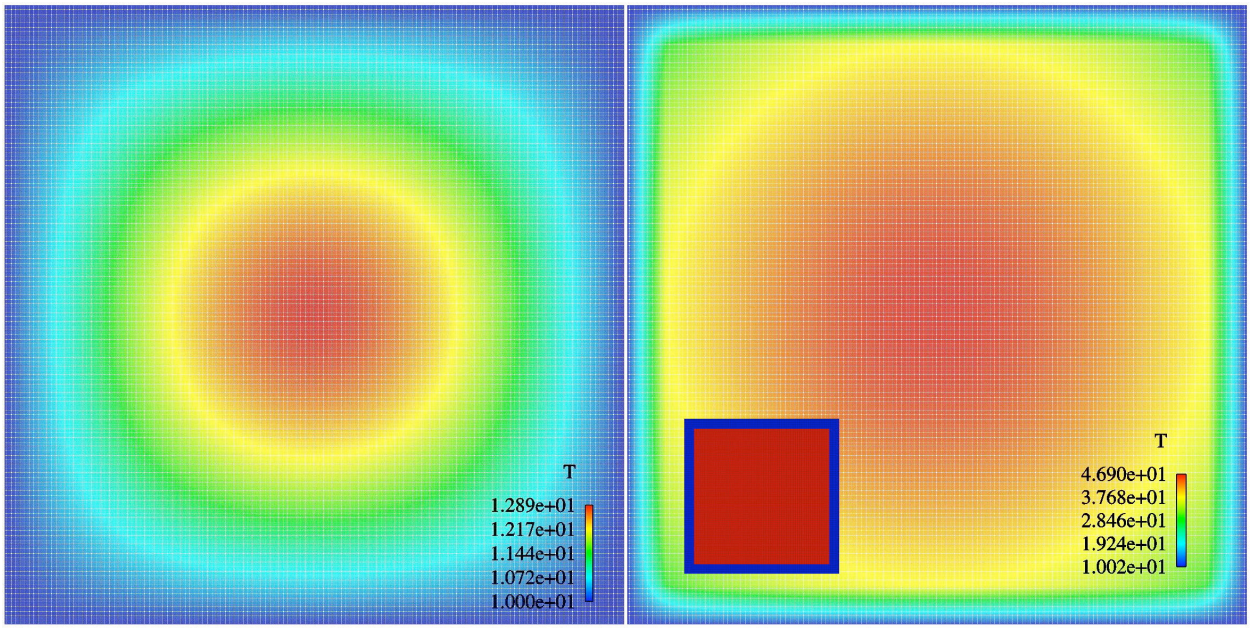

Figure 8. The Temperatures at t = 200 in two two-dimensional simulations. 512x512 grid cells are used. There is only one material of heat conductivity 0.001 in the left image, but in the right image there is another material of the conductivity (0.0001) in the outer layer shown in the picture-in-picture. The thickness of the second material is 0.0625.

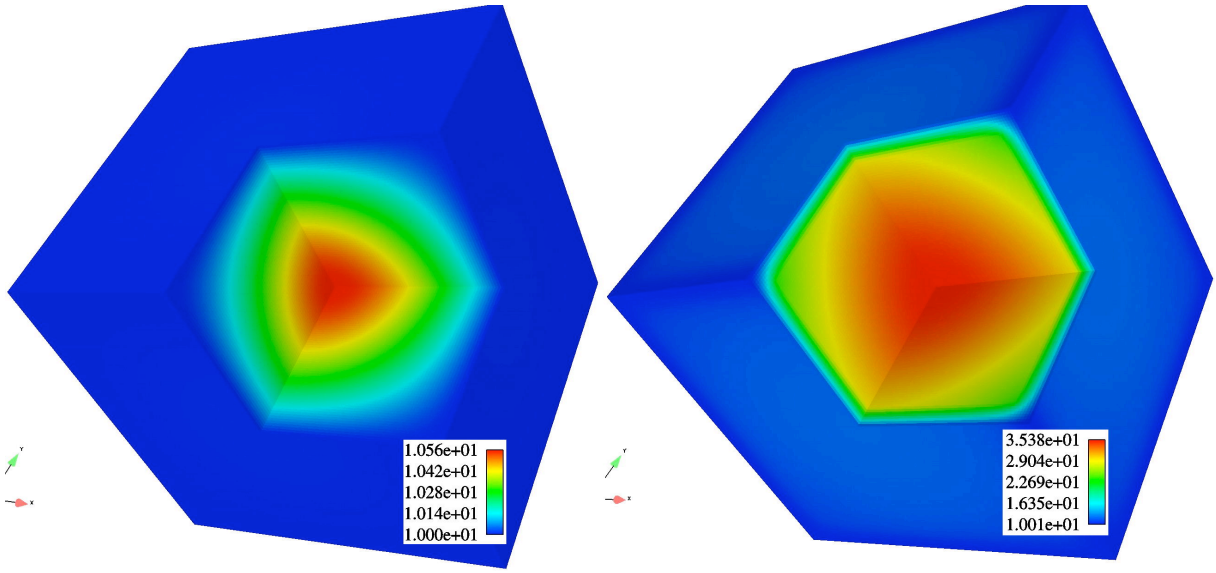

Figure 9. The temperature at t = 200 in two three-dimensional simulations. There is only one material of λ (= 0.001) in the left image, and in the right image there is the second material of λ (= 0.0001) at the outer layer of the simultion domain. The thickness of the second material is 0.0625.

The next four examples are simulations on two- and three-dimensional simulation domain with fixed temperature 10 at the boundaries. Initial temperature is 100 everywhere. The simulation domains are 1×1

or $1\times1\times1$ with 512x512 in two-dimensions and 128x128x128 in three-dimension. The left image in Fig.8 shows the distribution of T at time t = 200. In this simulation, the heat conductivity λ is 0.001 everywhere. The image at the right in Fig.8 shows the result at the same time when the outer layer of the simulation domain is another material of λ (=0.0001), which is shown in the picture-in-picture in the image.

The two three-dimensional simulations are the extension of the two-dimensional simulations described above. There are 128x128x128 grid cells in the simulation domains. Figure 9 displays the temperature T at t = 200 with single material (the left image) and two materials (the right image).

8. Conclusions

We have developed a numerical scheme for heat conduction in systems with very different materials. The schemes are second order accurate in both space and time. When the time step is very large, the scheme gives the correct steady states. The effective thermal conductivity for an interface of any two materials is derived in this paper based on physics principles. The scheme may correctly treat any number of materials with arbitrarily different materials. A very efficient iterative approach to solve the set of algebraic equations arising from the implicitness of the scheme is developed through the multigrid method. The features of the schemes are demonstrated through numerical examples in one-, two-, and three-dimensional situations.

9. Acknowledgments

We greatly appreciate the help from our teacher Lee Goodwin, and our mentor William Dai for their help in the project definition, direction in numerical methods for heat conduction, and the development of computer codes.

References

[1] J. Crank and P. Nicolson, Apractical method for numerical evaluation of solution of partical differential equations of the heat conduction type, proc. Camb. Phil. Soc. 43, 50 (1947).
[2] M. Jakob, Heat Transfer, Vol 1 (Wiley, New York, 1949).
[3] W. Wasow and G. Forsythe, Finite-difference Methods for Partial Differential Equations (Wiley, New York, 1960).
[4] D.M. Young, Iterative Solution of Large Linear Systems (Academic Press, New York, 1971).
[5] R.E. Alcouffe, A. Brandt, E. Dendy, and J. Painter, the multigrid method for diffusion equation with strong discontinuous coefficients, SIAM J. Sci. Stat. Comput. 2, 430 (1981).
[6] A. Brandt, Multi-level adaptive solution to boundary value problems, Math. Comput. 32, 333 (1977).
[7] P. Wesseling, Theroretical and practical aspects of a multigrid method, SIAM J. Sci. Comput. 3, 387 (1982).
[8] W. Hackbusch, Multi-grid methods and applications (Springer, Berlin, 1985).
[9] B.A. Fryxell, P.R. Woodward, P. Colella, and K.-H. Winkler, An implicit-explicit hybrid method for Lagrangian hydrodynamics, J. Comput. Phys. 63, 283 (1986).
[10] Y. Jaluria and K. E. Torrance, Computational Heat Transfer (Spring-Verlag, Berlin, 1986).
[11] S. F. McComick, Multigrid methods (Frontier in Applied Mathematics), Vol.3 (SIAM, Philadelphia, 1989).
[12] W. Dai and P. R. Woodward, Iterative implementation of an implicit-explicit hybrid scheme for hydrodynamics, J. Compt. Phys. 124, 217 (1996).

Appendix: Computer Source Codes

```
/********************** heat.h ******************************/
#ifndef MULTIGRID_H
#define MULTIGRID_H
#ifdef __cplusplus
extern "C" {
#endif
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
             ************************************************
             *****        THIS CODE WAS WRITTEN BY        *****
             *****            Edward J. DAI               *****
             *****            Aidan Bradbury              *****
             *****              April 2010                *****
             *****         ALL RIGHTS RESERVED            *****
             ************************************************
 ********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <assert.h>
#include <sys/time.h>
#include <time.h>
#include <assert.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>

#define MIN(a,b)    ((a) < (b) ? (a) : (b))
#define MAX(a,b)    ((a) > (b) ? (a) : (b))

struct H_Mesh;
```

```
struct H_Mesh {
       int level;          /* 0 means the finest mesh in multigrid */
       int sizes[3];       /* the number of cells in each dimension */
       double coordl[3];   /* lower bounds of simulation domain     */
       double coordr[3];   /* upper bounds of simulation domain     */

       /* old temperature, exist only for finest mesh */
       double *t1d;        /* 1D temperature, valid only for 1D case */
       double **t2d;       /* 2D temperature, valid only for 2D case */
       double ***t3d;      /* 3D temperature, valid only for 3D case */

       /* temperature after one time step */
       double *tn1d;        /* 1D temperature, valid only for 1D case */
       double **tn2d;       /* 2D temperature, valid only for 2D case */
       double ***tn3d;      /* 3D temperature, valid only for 3D case */

       /* temperature after a half time step */
       double *th1d;        /* 1D temperature, valid only for 1D case */
       double **th2d;       /* 2D temperature, valid only for 2D case */
       double ***th3d;      /* 3D temperature, valid only for 3D case */

       /* source terms for one time step */
       double *sn1d;        /* 1D source, valid only for 1D case */
       double **sn2d;       /* 2D source, valid only for 2D case */
       double ***sn3d;      /* 3D source, valid only for 3D case */

       /* source terms for half time step */
       double *sh1d;        /* 1D source, valid only for 1D case */
       double **sh2d;       /* 2D source, valid only for 2D case */
       double ***sh3d;      /* 3D source, valid only for 3D case */

       /* residues at one time step */
       double *rn1d;        /* 1D residue, valid only for 1D case */
       double **rn2d;       /* 2D residue, valid only for 2D case */
       double ***rn3d;      /* 3D residue, valid only for 3D case */
```

```
        /* residues at the half time step */
        double *rh1d;        /* 1D residue, valid only for 1D case */
        double **rh2d;       /* 2D residue, valid only for 2D case */
        double ***rh3d;      /* 3D residue, valid only for 3D case */

        /* heat conductivity */
        double *c1d;        /* 1D conductivity, valid only for 1D case */
        double **c2d;       /* 2D conductivity, valid only for 2D case */
        double ***c3d;      /* 3D conductivity, valid only for 3D case */

        double *cx1d, **cx2d, ***cx3d;   /* working array lambdax */
        double **cy2d, ***cy3d;          /* working array lambday */
        double ***cz3d;                  /* working array lambdaz */

        struct H_Mesh *parent;   /* parent mesh if not null */
        struct H_Mesh *child;    /* child mesh if not null  */
};
typedef struct H_Mesh H_Mesh;

enum H_BdryType {
     h_fixed      = 1,     /* fixed boundary condition */
     h_continued  = 2,     /* continuation bdoundary condition */
     h_not_bdy    = 10
 };
typedef enum H_BdryType H_BdryType;

struct H_BdryTemp {
        double tl, tr;    /* x direction, left and right */
        double tn, tf;    /* y direction, near and far   */
        double tb, tt;    /* z direction, bottom and top */
};
typedef struct H_BdryTemp H_BdryTemp;

#ifdef __cplusplus
}
```

```
#endif

#endif
```

/*---------- source code----------*/

```
#include "heat.h"
#include "mio.h"

const int dim = 3;
const int if_multigrid = 1;

double tbdry[] = {10, 10.0, 10.0, 10.0, 10.0, 10.0};  /* tl, tr, tn, tf, tb, tt */
double zeros[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
H_BdryType btypel = h_fixed;  /* x0 side boundary condition */
H_BdryType btyper = h_fixed;  /* x1 side boundary condition */
H_BdryType btypen = h_fixed;  /* y0 side boundary condition if dim >= 2 */
H_BdryType btypef = h_fixed;  /* y1 side boundary condition if dim >= 2 */
H_BdryType btypeb = h_fixed;  /* z0 side boundary condition if dim == 3 */
H_BdryType btypet = h_fixed;  /* z1 side boundary condition if dim == 3 */

double t_initial = 100.0;      /* initial temperature  */
double coef_air  = 10.0;       /* heat conductivity distribution */
double coef_wall = 10.0;       /* wall heat conductivity         */

double gcoordl[] = {0.0, 0.0, 0.0};   /* the lower ends of simulation domain */
double gcoordr[] = {1.0, 1.0, 1.0};   /* the high ends of simulation domain  */
int gsizes[] = {128, 128, 128};    /* sizes for finest mesh in multigrid  */
int nbdy = 1;                         /* boundary layer beyond gsizes        */

// one material
int nmat = 1;
double coef_array[] = {0.001};
double mat_coordl_array[1][3] = {{-2.0, -2.0, -2.0}};
```

```
double mat_coordr_array[1][3] = {{12.0, 12.0, 12.0}};
// two materials
/****
int nmat = 2;
double coef_array[] = {0.0001, 0.001};
double mat_coordl_array[3][3] = {{-1.0, -1.0, -1.0}, {0.0625, 0.0625, 0.0625}};
double mat_coordr_array[3][3] = {{ 2.0,  2.0,  2.0}, {0.9375, 0.9375, 0.9375}};
***/
/******
// three materials
int nmat = 2;
double coef_array[] = {0.001, 100.0, 1.0};
double mat_coordl_array[3][3] = {{-1.0, -1.0, -1.0}, {3.0, 3.0,  3.0}, {4.0, 4.0,
4.0}};
double mat_coordr_array[3][3] = {{11.0, 11.0, 11.0}, {7.0, 7.0,  7.0}, {6.0, 6.0,
6.0}};
****/

double coefbydc2;                   /* coef_max/(dc[0] * dc[0]) */
double courant;                     /* dt * coef_max/(dc[0] * dc[0]) */
double err_initial = 0.0;           /* Error obtained with the initial guess */

int niter  = 20000;                 /* max number of iteration */
int niterf = 6;                     /* the number of iteration on fine grid */
int niterc = 6;                     /* the number of iteration on coarse grids */
double afine = 0.0;                 /* accuracy on fine grid */
double acoar = 0.0;                 /* accuracy on coarse grids */
double small = 1.0e-10;


int    niter_max = 256;             /* max number of iteration for move_down */
double accuracy  = 1.0e-06;         /* accuracy */
double dt    = 0.1;                 /* time step          */
double tstop = 200.0;               /* time to terminate  */
int    nstop = 1000000000;          /* time steps to terminate */
```

```
double dtdump = 1.0;                  /* dump frequency */
int    ndump  = 100000000;            /* dump frequency in cycle */

clock_t t0_in_timer;        /* starting time for a timer */
clock_t t1_in_timer;        /* ending time of the timer  */

int update(H_Mesh *m);

int move_dn(H_Mesh *m, double dt, int niter_allowed, double *err, int *niter_already);
int map_dn1d(H_Mesh *m);
int map_dn2d(H_Mesh *m);
int map_dn3d(H_Mesh *m);
int move_up(H_Mesh *m, double dt, int niter_allowed, double *err, int *niter_already);
int map_up1d(H_Mesh *parent, double dt);
int map_up2d(H_Mesh *parent, double dt);
int map_up3d(H_Mesh *parent, double dt);
int heat_init(H_Mesh *mesh, double *gcoordl, double *gcoordr, int *sizes);
int set_coef_fr_parent(H_Mesh *child);
int initial_guess(H_Mesh *mesh);
int initial_guess0(H_Mesh *mesh);
int check_ifcoarse(H_Mesh *mesh, int *ifcoarse);
int dump(H_Mesh *m, double t, int idump);

int set_T_bdry1d(H_Mesh *m);
int set_cgc_bdry1d(double   *cgch,   double *cgcn, int *sizes);
int set_cgc_bdry2d(double  **cgch, double  **cgcn, int *sizes);
int set_cgc_bdry3d(double ***cgch, double ***cgcn, int *sizes);

int multigrid(H_Mesh *m, double dt);
int solver1d(H_Mesh *m, double dt, int niter_allowed, double accuracy_req, double
*err, int *niter_already);
int solver2d(H_Mesh *m, double dt, int niter_allowed, double accuracy_req, double
*err, int *niter_already);
int solver3d(H_Mesh *m, double dt, int niter_allowed, double accuracy_req, double
*err, int *niter_already);
int set_bdry(H_Mesh *m, double *bdry);
```

```
int set_heat_coef(H_Mesh *m);
int set_initial_temp(H_Mesh *m);
int set_gause_1d(H_Mesh *m);

int set_mesh(double *coordl, double *coordr, int *gsizes, H_Mesh *m);
int init_mesh(H_Mesh *m);
int set_2d_form(int *sizes, int nbdy, double ***data2d);
int set_3d_form(int *sizes, int nbdy, double ****data3d);

void err_msg(char *msg);

int main(int argc, char **argv)
{
    double t, tdump;
    int ncycle, todump, idump;
    H_Mesh mesh;

    t = 0;
    ncycle = 0;
    idump  = 0;
    tdump  = 0;
    todump = ndump;

/** set initial mesh, boundary condition, materials, and temperature **/

    heat_init(&mesh, gcoordl, gcoordr, gsizes);

/** This is to test accuracy of the method
    set_gause_1d(&mesh);
***/
    /* dump initial temperature */
    dump(&mesh, t, idump);
    idump++;

    while ((t < tstop) && (ncycle < nstop) ) {
        courant = dt * coefbydc2;
```

```
        printf("ncycle = %d, t = %e, courant = %e\n", ncycle, t, courant);
        /* Initial guess of iterative solver */
        initial_guess(&mesh);
        t0_in_timer = clock();;
        /* run multigrid method */
        multigrid(&mesh, dt);
        /* update temperature to t = t + dt */
        update(&mesh);

        t += dt;
        tdump += dt;
        todump--;
        ncycle++;

        if ((tdump >= dtdump) || (todump <= 0)) {
            /* dump temperature */
            dump(&mesh, t, idump);
            tdump = 0.0;
            todump = ndump;
            idump++;
        }
    }
    return 0;
}

int multigrid(H_Mesh *m, double dt)
{
/** This function run multigrid method to solve the system of algebraic equations */

    int niter_already = 0;
    double err;

    niter_already = 0;
    err = 1.0;
    while (err > accuracy) {
       /* move down of V cycle of multigrid method */
```

```
        move_dn(m, dt, niter_max, &err, &niter_already);
        if (if_multigrid) {
            /* move up of the V cycle if multigrid method */
            move_up(m, dt, niter_max, &err, &niter_already);
        }
    }
    return 0;
}

int move_dn(H_Mesh *m, double dt, int niter_allowed, double *err, int *niter_already)
{
   /* move done in the V cyle of multigrid method */

    H_Mesh *mesh, *parent, *child;

    assert(m && err);
    mesh = m;
    if (dim == 1) {
        while (mesh) {
            parent = mesh->parent;
            if (parent) {
                map_dn1d(parent);
            }
            if (mesh->parent == NULL) {
                solver1d(mesh, dt, niterf, afine, err, niter_already);
            }
            else {
                initial_guess0(mesh);   /* initial guess of correction */
                solver1d(mesh, dt, niterc, acoar, err, niter_already);
            }
            child = mesh->child;
            mesh  = child;
        }
    }
    else if (dim == 2) {
        while (mesh) {
```

```
            parent = mesh->parent;
            if (parent) {
                map_dn2d(parent);
            }
            if (mesh->parent == NULL) {
                solver2d(mesh, dt, niterf, afine, err, niter_already);
            }
            else {
                initial_guess0(mesh); /* initial guess of correction */
                solver2d(mesh, dt, niterc, acoar, err, niter_already);
            }
            child = mesh->child;
            mesh  = child;
        }
    }
    else if (dim == 3) {
        while (mesh) {
            parent = mesh->parent;
            if (parent) {
                map_dn3d(parent);
            }
            if (mesh->parent == NULL) {
                solver3d(mesh, dt, niterf, afine, err, niter_already);
            }
            else {
                initial_guess0(mesh); /* initial guess of correction */
                solver3d(mesh, dt, niterc, acoar, err, niter_already);
            }
            child = mesh->child;
            mesh = child;
        }
    }
    return 0;
 }

int move_up(H_Mesh *m, double dt, int niter_allowed, double *err, int *niter_already)
```

```
{
  /* Move up of the V cycle in multigrid method */

    int mylevel;
    H_Mesh *parent, *mesh, *child;

    assert(m && err);
    mylevel = m->level;
    mesh = m;
    while (mesh->child) {
       mesh = mesh->child;
    }
    parent = mesh->parent;
    if (dim == 1) {
        while (parent && (mesh->level > mylevel) ) {
            map_up1d(parent, dt);
            if (parent->parent == NULL) {
                solver1d(parent, dt, niterf, afine, err, niter_already);
            }
            else {
                solver1d(parent, dt, niterc, acoar, err, niter_already);
            }
            if (*err > accuracy) {
                /* If not satisfied, do a sub-V cycle */
                move_dn(parent->child, dt, niter_allowed, err, niter_already);
                move_up(parent, dt, niter_allowed, err, niter_already);
            }
            mesh   = parent;
            parent = parent->parent;
        }
    }
    else if (dim == 2) {
        while (parent && (mesh->level > mylevel)) {
            map_up2d(parent, dt);
            if (parent->parent == NULL) {
                solver2d(parent, dt, niterf, afine, err, niter_already);
```

```
            }
            else {
                solver2d(parent, dt, niterc, acoar, err, niter_already);
            }
            if (*err > accuracy) {
                /* If not satisfied, do a sub-V cycle */
                move_dn(parent->child, dt, niter_allowed, err, niter_already);
                move_up(parent, dt, niter_allowed, err, niter_already);
            }
            mesh   = parent;
            parent = parent->parent;
        }
    }
    else if (dim == 3) {
        while (parent && (mesh->level > mylevel)) {
            map_up3d(parent, dt);
            if (parent->parent == NULL) {
                solver3d(parent, dt, niterf, afine, err, niter_already);
            }
            else {
                solver3d(parent, dt, niterc, acoar, err, niter_already);
            }
            if (*err > accuracy) {
                /* If not satisfied, do a sub-V cycle */
                move_dn(parent->child, dt, niter_allowed, err, niter_already);
                move_up(parent, dt, niter_allowed, err, niter_already);
            }
            mesh   = parent;
            parent = parent->parent;
        }
    }
    return 0;
 }

int heat_init(H_Mesh *mesh, double *gcoordl, double *gcoordr, int *sizes)
{
```

```
/*  Set mesh, materials, and initial tempertaure */

    int d, to_coarsen;
    int hsizes[3];
    H_Mesh *parent, *child;

    set_mesh(gcoordl, gcoordr, sizes, mesh);
    set_heat_coef(mesh);    /* set heat coefficients */
    set_initial_temp(mesh); /* set initial temperature */
    set_bdry(mesh, tbdry);
    if (if_multigrid == 0) return 0;

    parent = mesh;
    check_ifcoarse(parent, &to_coarsen);   /* check if to coarsen */
    while (to_coarsen) {  /* set a child (coarse) mesh  */
        for (d = 0; d < dim; d++) {
            hsizes[d] = parent->sizes[d] / 2;
        }
        child = (H_Mesh *) malloc(sizeof(H_Mesh));
        set_mesh(gcoordl, gcoordr, hsizes, child);   /* set mesh */
        child->parent = parent;
        child->level = parent->level + 1;
        parent->child = child;
        set_coef_fr_parent(child);    /* set heat coefficients */
        parent = child;
        check_ifcoarse(parent, &to_coarsen);   /* check if to coarsen */
        if (hsizes[0] <= 4) {
            to_coarsen = 0;
        }
    }
    return 0;
}

int map_up1d(H_Mesh *parent, double dt)
{
/*  map temperature from a coase grid to a fine grid */
```

```
    int *sizes, i0, i, ic;
    double *th, *tn, *thc, *tnc;
    H_Mesh *child;

    assert(parent);
    sizes = parent->sizes;
    th = parent->th1d;
    tn = parent->tn1d;
    child = parent->child;
    assert(child);
    thc = child->th1d;
    tnc = child->tn1d;
    for (i0 = 0; i0 < sizes[0]; i0++) {
        i  = nbdy + i0;
        ic = nbdy + i0/2;
        th[i] -= thc[ic];
        tn[i] -= tnc[ic];
    }
    return 0;
 }

int map_up2d(H_Mesh *parent, double dt)
{
/*  map temperature from a coase grid to a fine grid */

    int *sizes, i0, i, ic, j0, j, jc;
    double **th, **tn, **thc, **tnc;
    H_Mesh *child;

    assert(parent);
    sizes = parent->sizes;
    th = parent->th2d;
    tn = parent->tn2d;
    child = parent->child;
    assert(child);
```

```
    thc = child->th2d;
    tnc = child->tn2d;
    for (j0 = 0; j0 < sizes[1]; j0++) {
        j = nbdy + j0;
        jc = nbdy + j0/2;
        for (i0 = 0; i0 < sizes[0]; i0++) {
            i = nbdy + i0;
            ic = nbdy + i0/2;
            th[j][i] -= thc[jc][ic];
            tn[j][i] -= tnc[jc][ic];
        }
    }
    return 0;
 }

int map_up3d(H_Mesh *parent, double dt)
{
/*  map temperature from a coase grid to a fine grid */

    int *sizes, i0, i, ic, j0, j, jc, k0, k, kc;
    double ***th, ***tn, ***thc, ***tnc;
    H_Mesh *child;

    assert(parent);
    sizes = parent->sizes;
    th = parent->th3d;
    tn = parent->tn3d;
    child = parent->child;
    assert(child);
    thc = child->th3d;
    tnc = child->tn3d;
    for (k0 = 0; k0 < sizes[2]; k0++) {
        k  = nbdy + k0;
        kc = nbdy + k0/2;
        for (j0 = 0; j0 < sizes[1]; j0++) {
            j  = nbdy + j0;
```

```
            jc = nbdy + j0/2;
            for (i0 = 0; i0 < sizes[0]; i0++) {
                i  = nbdy + i0;
                ic = nbdy + i0/2;
                th[k][j][i] -= thc[kc][jc][ic];
                tn[k][j][i] -= tnc[kc][jc][ic];
            }
        }
    }
    return 0;
 }


int check_ifcoarse(H_Mesh *mesh, int *ifcoarse)
{
/*  check whether the mesh could be further coarsen if not mix material interafces */

    int *sizes, hsizes[3], d;
    int i0, ip0, ip1, ip, j0, jp0, jp1, jp, k0, kp0, kp1, kp;
    double *c1d, **c2d, ***c3d, value;

    *ifcoarse = 1;
    sizes = mesh->sizes;
    for (d = 0; d < dim; d++) {
        if (sizes[d] == 1) {
            *ifcoarse = 0;
            break;
        }
    }
    if (!(*ifcoarse)) return 0;

    for (d = 0; d < dim; d++) {
        hsizes[d] = sizes[d] / 2;
    }
    if (dim == 1) {
        c1d = mesh->c1d;
```

```
    for (i0 = 0; i0 < hsizes[0]; i0++) {
        ip0 = i0 + i0 + nbdy;
        ip1 = ip0 + 2;
        value = c1d[ip0];
        for (ip = ip0; ip < ip1; ip++) {
            if (value != c1d[ip]) {
                *ifcoarse = 0;
                return 0;
            }
        }
    }
}
else if (dim == 2) {
    c2d = mesh->c2d;
    for (j0 = 0; j0 < hsizes[1]; j0++) {
        jp0 = j0 + j0 + nbdy;
        jp1 = jp0 + 2;
        for (i0 = 0; i0 < hsizes[0]; i0++) {
            ip0 = i0 + i0 + nbdy;
            ip1 = ip0 + 2;
            value = c2d[jp0][ip0];
            for (jp = jp0; jp < jp1; jp++) {
                for (ip = ip0; ip < ip1; ip++) {
                    if (value != c2d[jp][ip]) {
                       *ifcoarse = 0;
                       return 0;
                    }
                }
            }
        }
    }
}
else if (dim == 3) {
    c3d = mesh->c3d;
    for (k0 = 0; k0 < hsizes[2]; k0++) {
        kp0 = k0 + k0 + nbdy;
```

```
            kp1 = kp0 + 2;
            for (j0 = 0; j0 < hsizes[1]; j0++) {
                jp0 = j0 + j0 + nbdy;
                jp1 = jp0 + 2;
                for (i0 = 0; i0 < hsizes[0]; i0++) {
                    ip0 = i0 + i0 + nbdy;
                    ip1 = ip0 + 2;
                    value = c3d[kp0][jp0][ip0];
                    for (kp = kp0; kp < kp1; kp++) {
                        for (jp = jp0; jp < jp1; jp++) {
                            for (ip = ip0; ip < ip1; ip++) {
                                if (value != c3d[kp][jp][ip]) {
                                   *ifcoarse = 0;
                                   return 0;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return 0;
}

int update(H_Mesh *mesh)
{
 /* assign T_new to T_old */

    int i, j, k, *sizes;
    double *t1d, *th1d, *tn1d, **t2d, **th2d, **tn2d;
    double ***t3d, ***th3d, ***tn3d;

    assert(mesh);
    assert(!(mesh->parent));
    sizes = mesh->sizes;
```

```
    if (dim == 1) {
        th1d = mesh->th1d;
        tn1d = mesh->tn1d;
        t1d  = mesh->t1d;
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            t1d[i] = tn1d[i];
            th1d[i] = tn1d[i];
        }
    }
    else if (dim == 2) {
        th2d = mesh->th2d;
        tn2d = mesh->tn2d;
        t2d  = mesh->t2d;
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                t2d[j][i] = tn2d[j][i];
                th2d[j][i] = tn2d[j][i];
            }
        }
    }
    else if (dim == 3) {
        th3d = mesh->th3d;
        tn3d = mesh->tn3d;
        t3d  = mesh->t3d;
        for (k = 0; k < sizes[2] + nbdy + nbdy; k++) {
            for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
                for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                    t3d[k][j][i] = tn3d[k][j][i];
                    th3d[k][j][i] = tn3d[k][j][i];
                }
            }
        }
    }
    return 0;
}
```

```
int initial_guess(H_Mesh *mesh)
{
/*  initial guess for temperature in interative solver */

    int i, j, k, *sizes;
    double *t1d, *th1d, *tn1d, **t2d, **th2d, **tn2d;
    double ***t3d, ***th3d, ***tn3d;
    H_Mesh *child;

    assert(mesh);
    assert(!(mesh->parent));
    sizes = mesh->sizes;

    if (dim == 1) {
        th1d = mesh->th1d;
        tn1d = mesh->tn1d;
        t1d  = mesh->t1d;
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            tn1d[i] = t1d[i];
            th1d[i] = t1d[i];
        }
    }
    else if (dim == 2) {
        th2d = mesh->th2d;
        tn2d = mesh->tn2d;
        t2d  = mesh->t2d;
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                tn2d[j][i] = t2d[j][i];
                th2d[j][i] = t2d[j][i];
            }
        }
    }
    else if (dim == 3) {
        th3d = mesh->th3d;
```

```
        tn3d = mesh->tn3d;
        t3d  = mesh->t3d;
        for (k = 0; k < sizes[2] + nbdy + nbdy; k++) {
            for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
                for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                    tn3d[k][j][i] = t3d[k][j][i];
                    th3d[k][j][i] = t3d[k][j][i];
                }
            }
        }
    }
    child = mesh->child;
    while (child) {
        initial_guess0(child);
        child = child->child;
    }
    return 0;
}

int initial_guess0(H_Mesh *mesh)
{
/*  Initial guess for coorections in iterative solver */
    int i, j, k, *sizes;
    double *t1d, *th1d, *tn1d, **t2d, **th2d, **tn2d;
    double ***t3d, ***th3d, ***tn3d;

    sizes = mesh->sizes;
    if (dim == 1) {
        th1d = mesh->th1d;
        tn1d = mesh->tn1d;
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            tn1d[i] = 0.0;
            th1d[i] = 0.0;
        }
    }
    else if (dim == 2) {
```

```
        th2d = mesh->th2d;
        tn2d = mesh->tn2d;
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                tn2d[j][i] = 0.0;
                th2d[j][i] = 0.0;
            }
        }
    }
    else if (dim == 3) {
        th3d = mesh->th3d;
        tn3d = mesh->tn3d;
        for (k = 0; k < sizes[2] + nbdy + nbdy; k++) {
            for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
                for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                    tn3d[k][j][i] = 0.0;
                    th3d[k][j][i] = 0.0;
                }
            }
        }
    }
    return 0;
 }

int map_dn1d(H_Mesh *m)
{
/*  Map residues of parent to the source term of child in 1D */

    int i0, i, i00, i11, ip0, ip;
    int *sizes;
    double *rh, *rn, *sh, *sn, vn, vh;
    H_Mesh *child;

    rh = m->rh1d;
    rn = m->rn1d;
```

```
    child = m->child;
    assert(child);
    sh    = child->sh1d;
    sn    = child->sn1d;
    sizes = child->sizes;
    for (i0 = 0; i0 < sizes[0]; i0++) {
        i = nbdy + i0;
        i00 = i0  + i0;
        i11 = i00 + 2;
        vn = 0.0;
        vh = 0.0;
        for (ip0 = i00; ip0 < i11; ip0++) {
            ip = nbdy + ip0;
            vn += rn[ip];
            vh += rh[ip];
        }
        sh[i] = 0.5 * vh;
        sn[i] = 0.5 * vn;
    }
    return 0;
}

int map_dn2d(H_Mesh *m)
{
/*  Map residues of parent to the source term of child in 2D */

    int i0, i, i00, i11, ip0, ip, j0, j, j00, j11, jp0, jp;
    int *sizes;
    double **rh, **rn, **sh, **sn, vn, vh;
    H_Mesh *child;

    rh = m->rh2d;
    rn = m->rn2d;
    child = m->child;
    assert(child);
    sh    = child->sh2d;
```

```
    sn    = child->sn2d;
    sizes = child->sizes;
    for (j0 = 0; j0 < sizes[0]; j0++) {
        j = nbdy + j0;
        j00 = j0  + j0;
        j11 = j00 + 2;
        for (i0 = 0; i0 < sizes[0]; i0++) {
            i = nbdy + i0;
            i00 = i0  + i0;
            i11 = i00 + 2;

            vn = 0.0;
            vh = 0.0;
            for (jp0 = j00; jp0 < j11; jp0++) {
                jp = nbdy + jp0;
                for (ip0 = i00; ip0 < i11; ip0++) {
                    ip  = nbdy + ip0;
                    vn += rn[jp][ip];
                    vh += rh[jp][ip];
                }
            }
            sh[j][i] = 0.25 * vh;
            sn[j][i] = 0.25 * vn;
        }
    }
    return 0;
}

int map_dn3d(H_Mesh *m)
{
/*  Map residues of parent to the source term of child in 3D */

    int i0, i, i00, i11, ip0, ip, j0, j, j00, j11, jp0, jp;
    int k0, k, k00, k11, kp0, kp;
    int *sizes;
    double ***rh, ***rn, ***sh, ***sn, vn, vh;
```

```
H_Mesh *child;


rh = m->rh3d;
rn = m->rn3d;
child = m->child;
assert(child);
sh    = child->sh3d;
sn    = child->sn3d;
sizes = child->sizes;
for (k0 = 0; k0 < sizes[0]; k0++) {
    k = nbdy + k0;
    k00 = k0  + k0;
    k11 = k00 + 2;
    for (j0 = 0; j0 < sizes[0]; j0++) {
        j = nbdy + j0;
        j00 = j0  + j0;
        j11 = j00 + 2;
        for (i0 = 0; i0 < sizes[0]; i0++) {
            i = nbdy + i0;
            i00 = i0  + i0;
            i11 = i00 + 2;


            vn = 0.0;
            vh = 0.0;
            for (kp0 = k00; kp0 < k11; kp0++) {
                kp = nbdy + kp0;
                for (jp0 = j00; jp0 < j11; jp0++) {
                    jp = nbdy + jp0;
                    for (ip0 = i00; ip0 < i11; ip0++) {
                        ip = nbdy + ip0;
                        vn += rn[kp][jp][ip];
                        vh += rh[kp][jp][ip];
                    }
                }
            }
            sh[k][j][i] = 0.125 * vh;
```

```
                sn[k][j][i] = 0.125 * vn;
            }
        }
    }
    return 0;
}

int solver1d(H_Mesh *m, double dt, int niter_allowed, double accuracy_req, double
*err_out, int *niter_already)
{
/*  iterative solver in one fixed grid for 1D */

    int i0, i, iter, clr;
    int *sizes;
    double err0, dc, dtbydc2[3], tmp[3];
    double dtime_in_sec, ddn, ddh, factor, srcdt, etadt, etadtbyc, srcn, srch;
    double courinv, courinv2,  aa, ainv, qn, qh, tni, thi;
    double *coordl, *coordr;
    double *t, *tn, *th, *sn, *sh, *rn, *rh, *cx;

    courinv = 1.0/courant;
    courinv2 = courinv * courinv;

    coordl = m->coordl;
    coordr = m->coordr;
    sizes  = m->sizes;
    for (i = 0; i < dim; i++) {
        dc = (coordr[i] - coordl[i])/(double) sizes[i];
        dtbydc2[i] = dt/(dc * dc);
        tmp[i] = dtbydc2[i] * courinv;
    }
    t   = m->t1d;
    tn  = m->tn1d;
    th  = m->th1d;
    sn  = m->sn1d;
    sh  = m->sh1d;
```

```
rn  = m->rn1d;
rh  = m->rh1d;

cx  = m->cx1d;
srcdt = 0.0;
if (m->parent == NULL) {
    for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
        sn[i] = 0.0;
        sh[i] = 0.0;
    }
}
else {
    for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
        t[i]  = 0.0;
    }
}
iter = 0;
while (iter < niter_allowed)  {
    if (m->parent) {
        set_cgc_bdry1d(tn, th, sizes);
    }
    else {
        set_T_bdry1d(m);
    }
    for (clr = 0; clr < 2; clr++) {
        for (i0 = clr; i0 < sizes[0]; i0 += 2) {
            i = i0 + nbdy;
            etadt = (cx[i] + cx[i+1]) * dtbydc2[0];
            ainv = 1.0 /(1.0 + 0.25 * etadt *(3.0 + etadt));
            qh   = dtbydc2[0] *(cx[i] * th[i-1] + cx[i+1] * th[i+1]);
            qn   = dtbydc2[0] *(cx[i] * tn[i-1] + cx[i+1] * tn[i+1]);
            srcn = (1.0 + 0.25 * etadt) * srcdt
                 + (1.0 + 0.75 * etadt) * sn[i] - etadt * sh[i];
            srch = (0.5 + 0.25 * etadt) * srcdt
                 + (0.25 * etadt * sn[i] + sh[i]);
            ddn = (1.0 - 0.25 * etadt) *t[i] + srcn + qh + 0.25 * etadt * qn;
```

```
                ddh = (1.0 + 0.25 * etadt) *t[i] + srch + (0.75 + 0.25 * etadt) * qh -
0.25 * qn;
                tni = ddn * ainv;
                thi = ddh * ainv;
                tn[i] = tni;
                th[i] = thi;
            }
        }
        *err_out = 0.0;
        for (i0 = 0; i0 < sizes[0]; i0++) {
            i = nbdy + i0;
            etadt = (cx[i] + cx[i+1]) * dtbydc2[0];
            qh   = dtbydc2[0] *(cx[i] * th[i-1] + cx[i+1] * th[i+1]);
            qn   = dtbydc2[0] *(cx[i] * tn[i-1] + cx[i+1] * tn[i+1]);
            rn[i] = tn[i] + etadt * th[i] - (t[i] + srcdt + qh + sn[i]);
            rh[i] = -0.25 * etadt * tn[i] + (1.0 + 0.75 * etadt) * th[i]
                  - (t[i] + 0.5 * srcdt + 0.75 * qh - 0.25 * qn + sh[i]);

            ainv  = 1.0/((1.0 + etadt) * (1.0 + tn[i]));
            *err_out = MAX(*err_out, MAX(fabs(rn[i] * ainv), fabs(rh[i] * ainv) ) );
        }
        if (iter == 0) {
            err0 = *err_out;
        }
        iter++;
        if (m->parent == NULL) {
            (*niter_already)++;
            t1_in_timer = clock();
            dtime_in_sec = (t1_in_timer - t0_in_timer)/(double)CLOCKS_PER_SEC;
        }
    }
    return 0;
 }

int solver2d(H_Mesh *m, double dt, int niter_allowed, double accuracy_req, double
*err_out, int *niter_already)
```

```
{
/*  iterative solver in one fixed grid for 2D */

    int clr, istart, i0, i, j0, j, iter;
    int *sizes;
    double dc, dtbydc2[3], tmp[3], err0;
    double dtime_in_sec, ddn, ddh, factor, srcdt, etadt, etadtbyc, srcn, srch;
    double courinv, courinv2,  aa, ainv, qn, qh, tni, thi;
    double *coordl, *coordr;
    double **t, **tn, **th, **sn, **sh, **rn, **rh, **cx, **cy;

    courinv = 1.0/courant;
    courinv2 = courinv * courinv;

    coordl = m->coordl;
    coordr = m->coordr;
    sizes  = m->sizes;
    for (i = 0; i < dim; i++) {
        dc = (coordr[i] - coordl[i])/(double) sizes[i];
        dtbydc2[i] = dt/(dc * dc);
        tmp[i] = dtbydc2[i] * courinv;
    }
    t   = m->t2d;
    tn  = m->tn2d;
    th  = m->th2d;
    sn  = m->sn2d;
    sh  = m->sh2d;
    rn  = m->rn2d;
    rh  = m->rh2d;
    cx  = m->cx2d;
    cy  = m->cy2d;

    factor = 0.5 / (double) (sizes[0] * sizes[1]);
    srcdt  = 0.0;

    iter = 0;
```

```
    while (iter < niter_allowed) {
        if (m->parent) {
            set_cgc_bdry2d(tn, th, sizes);
        }
        for (clr = 0; clr < 2; clr++) {
            istart = clr;
            for (j0 = 0; j0 < sizes[1]; j0++) {
                j = nbdy + j0;
                for (i0 = istart; i0 < sizes[0]; i0 += 2) {
                    i = nbdy + i0;
                    etadt = (cx[j][i] + cx[j][i+1]) * dtbydc2[0]
                          + (cy[j][i] + cy[j+1][i]) * dtbydc2[1];
                    ainv = 1.0 /(1.0 + 0.25 * etadt *(3.0 + etadt));
                    qh   = dtbydc2[0] *(cx[j][i] * th[j][i-1] + cx[j][i+1] *
th[j][i+1])
                         + dtbydc2[1] *(cy[j][i] * th[j-1][i] + cy[j+1][i] *
th[j+1][i]);
                    qn   = dtbydc2[0] *(cx[j][i] * tn[j][i-1] + cx[j][i+1] *
tn[j][i+1])
                         + dtbydc2[1] *(cy[j][i] * tn[j-1][i] + cy[j+1][i] *
tn[j+1][i]);
                    srcn = (1.0 + 0.25 * etadt) * srcdt
                         + (1.0 + 0.75 * etadt) * sn[j][i] - etadt * sh[j][i];
                    srch = (0.5 + 0.25 * etadt) * srcdt
                         + (0.25 * etadt * sn[j][i] + sh[j][i]);

                    ddn = (1.0 - 0.25 * etadt) * t[j][i] + srcn + qh + 0.25 * etadt *
qn;
                    ddh = (1.0 + 0.25 * etadt) * t[j][i] + srch + (0.75 + 0.25 *
etadt) * qh - 0.25 * qn;
                    tn[j][i] = ddn * ainv;
                    th[j][i] = ddh * ainv;
                }
                istart = 1 - istart;
            }
        }
```

```
        *err_out = 0.0;
        for (j0 = 0; j0 < sizes[1]; j0++) {
            j = nbdy + j0;
            for (i0 = 0; i0 < sizes[0]; i0++) {
                i = nbdy + i0;
                etadt = (cx[j][i] + cx[j][i+1]) * dtbydc2[0]
                      + (cy[j][i] + cy[j+1][i]) * dtbydc2[1];
                qh   = dtbydc2[0] *(cx[j][i] * th[j][i-1] + cx[j][i+1] * th[j][i+1])
                     + dtbydc2[1] *(cy[j][i] * th[j-1][i] + cy[j+1][i] * th[j+1][i]);
                qn   = dtbydc2[0] *(cx[j][i] * tn[j][i-1] + cx[j][i+1] * tn[j][i+1])
                     + dtbydc2[1] *(cy[j][i] * tn[j-1][i] + cy[j+1][i] * tn[j+1][i]);
                rn[j][i] = tn[j][i] + etadt * th[j][i] - (t[j][i] + srcdt + qh +
sn[j][i]);
                rh[j][i] = -0.25 * etadt * tn[j][i] + (1.0 + 0.75 * etadt) * th[j][i]
                         - (t[j][i] + 0.5 * srcdt + 0.75 * qh - 0.25 * qn + sh[j][i]);

                ainv  = 1.0/((1.0 + etadt) * (1.0 + tn[j][i]));
                *err_out = MAX(*err_out, MAX(fabs(rn[j][i] * ainv), fabs(rh[j][i] *
ainv) ) );
            }
        }
        if (iter == 0) {
            err0 = *err_out;
        }
        iter++;
        if (m->parent == NULL) {
            (*niter_already)++;
            t1_in_timer = clock();
            dtime_in_sec = (t1_in_timer - t0_in_timer)/(double)CLOCKS_PER_SEC;
        }
    }
    return 0;
 }

int solver3d(H_Mesh *m, double dt, int niter_allowed, double accuracy_req, double
*err_out, int *niter_already)
```

```
{
    int clr, i0, j0, k0, i, j, k, iter, istart0, istart;
    int *sizes;
    double dtime_in_sec, dc, dtbydc2[3], tmp[3];
    double err0, factor, srcdt, etadt, etadtbyc, srcn, srch;
    double courinv, courinv2,  aa, ainv, qn, qh, tni, thi;
    double *coordl, *coordr;
    double ***t, ***tn, ***th, ***cx, ***cy, ***cz;
    double ***sn, ***sh, ***rh, ***rn;

    courinv = 1.0/courant;
    courinv2 = courinv * courinv;

    coordl = m->coordl;
    coordr = m->coordr;
    sizes  = m->sizes;
    for (i = 0; i < dim; i++) {
        dc = (coordr[i] - coordl[i])/(double) sizes[i];
        dtbydc2[i] = dt/(dc * dc);
        tmp[i] = dtbydc2[i] * courinv;
    }
    t   = m->t3d;
    tn  = m->tn3d;
    th  = m->th3d;
    sn  = m->sn3d;
    sh  = m->sh3d;
    rn  = m->rn3d;
    rh  = m->rh3d;
    cx  = m->cx3d;
    cy  = m->cy3d;
    cz  = m->cz3d;

    factor = 0.5 / (double)(sizes[0] * sizes[1] * sizes[2]);
    srcdt  = 0.0;

    iter = 0;
```

```
    while (iter < niter_allowed) {
        if (m->parent) {
            set_cgc_bdry3d(tn, th, sizes);
        }
        for (clr = 0; clr < 2; clr++) {
            istart0 = clr;
            for (k0 = 0; k0 < sizes[2]; k0++) {
                k = nbdy + k0;
                istart = istart0;
                for (j0 = 0; j0 < sizes[1]; j0++) {
                    j = nbdy + j0;
                    for (i0 = istart; i0 < sizes[0]; i0 += 2) {
                        i = nbdy + i0;
                        etadt = (cx[k][j][i] + cx[k][j][i+1]) * dtbydc2[0]
                              + (cy[k][j][i] + cy[k][j+1][i]) * dtbydc2[1]
                              + (cz[k][j][i] + cz[k+1][j][i]) * dtbydc2[2];
                        ainv = 1.0 /(1.0 + 0.25 * etadt *(3.0 + etadt));
                        qh   = dtbydc2[0] *(cx[k][j][i] * th[k][j][i-1] +
cx[k][j][i+1] * th[k][j][i+1])
                             + dtbydc2[1] *(cy[k][j][i] * th[k][j-1][i] +
cy[k][j+1][i] * th[k][j+1][i])
                             + dtbydc2[2] *(cz[k][j][i] * th[k-1][j][i] +
cz[k+1][j][i] * th[k+1][j][i]);
                        qn   = dtbydc2[0] *(cx[k][j][i] * tn[k][j][i-1] +
cx[k][j][i+1] * tn[k][j][i+1])
                             + dtbydc2[1] *(cy[k][j][i] * tn[k][j-1][i] +
cy[k][j+1][i] * tn[k][j+1][i])
                             + dtbydc2[2] *(cz[k][j][i] * tn[k-1][j][i] +
cz[k+1][j][i] * tn[k+1][j][i]);
                        srcn = (1.0 + 0.25 * etadt) * srcdt
                             + (1.0 + 0.75 * etadt) * sn[k][j][i] - etadt *
sh[k][j][i];
                        srch = (0.5 + 0.25 * etadt) * srcdt
                             + (0.25 * etadt * sn[k][j][i] + sh[k][j][i]);
                        tni  = ((1.0 - 0.25 * etadt) * t[k][j][i] + srcn + qh + 0.25 *
etadt * qn) * ainv;
```

```
                    thi  = ((1.0 + 0.25 * etadt) * t[k][j][i] + srch
                         +  (0.75 + 0.25 * etadt) * qh - 0.25 * qn) * ainv;
                    tn[k][j][i] = tni;
                    th[k][j][i] = thi;
                }
                istart = 1 - istart;
            }
            istart0 = 1 - istart0;
        }
    }
    *err_out = 0.0;
    for (k0 = 0; k0 < sizes[2]; k0++) {
        k = nbdy + k0;
        for (j0 = 0; j0 < sizes[1]; j0++) {
            j = nbdy + j0;
            for (i0 = 0; i0 < sizes[0]; i0++) {
                i = nbdy + i0;
                etadt = (cx[k][j][i] + cx[k][j][i+1]) * dtbydc2[0]
                      + (cy[k][j][i] + cy[k][j+1][i]) * dtbydc2[1]
                      + (cz[k][j][i] + cz[k+1][j][i]) * dtbydc2[2];
                qh  = dtbydc2[0] *(cx[k][j][i] * th[k][j][i-1] + cx[k][j][i+1] *
th[k][j][i+1])
                    + dtbydc2[1] *(cy[k][j][i] * th[k][j-1][i] + cy[k][j+1][i] *
th[k][j+1][i])
                    + dtbydc2[2] *(cz[k][j][i] * th[k-1][j][i] + cz[k+1][j][i] *
th[k+1][j][i]);
                qn  = dtbydc2[0] *(cx[k][j][i] * tn[k][j][i-1] + cx[k][j][i+1] *
tn[k][j][i+1])
                    + dtbydc2[1] *(cy[k][j][i] * tn[k][j-1][i] + cy[k][j+1][i] *
tn[k][j+1][i])
                    + dtbydc2[2] *(cz[k][j][i] * tn[k-1][j][i] + cz[k+1][j][i] *
tn[k+1][j][i]);

                rn[k][j][i] = tn[k][j][i] + etadt * th[k][j][i] - (t[k][j][i] +
srcdt + qh + sn[k][j][i]);
```

```
                    rh[k][j][i] = -0.25 * etadt * tn[k][j][i] + (1.0 + 0.75 * etadt) *
th[k][j][i]
                                - (t[k][j][i] + 0.5 * srcdt + 0.75 * qh - 0.25 * qn +
sh[k][j][i]);

                    ainv  = 1.0/((1.0 + etadt) * (1.0 + tn[k][j][i]));
                    *err_out = MAX(*err_out, MAX(fabs(rn[k][j][i] * ainv),
fabs(rh[k][j][i] * ainv) ) );
                }
            }
        }
        if (iter == 0) {
            err0 = *err_out;
        }
        iter++;
        if (m->parent == NULL) {
            (*niter_already)++;
            t1_in_timer = clock();
            dtime_in_sec = (t1_in_timer - t0_in_timer)/(double)CLOCKS_PER_SEC;
//            printf("%5d  %e  %e\n", *niter_already, dtime_in_sec, *err_out);
        }
    }
    return 0;
 }

int set_bdry(H_Mesh *m, double *bdry)
{
/* set fixed boundary conditions */

   int i, i1, j, j1, k, k1;
   int *sizes;
   double *t1d, **t2d, ***t3d;

   sizes = m->sizes;
   if (dim == 1) {
       t1d = m->t1d;
```

```
    for (i = 0; i < nbdy; i++) {
        i1 = sizes[0] + nbdy + i;
        t1d[i]  = bdry[0];
        t1d[i1] = bdry[1];
    }
}
else if (dim == 2) {
    t2d = m->t2d;
    for (j = 0; j < nbdy; j++) {
        j1 = sizes[1] + nbdy + j;
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            t2d[j][i]  = bdry[2];
            t2d[j1][i] = bdry[3];
        }
    }
    for (j = 0; j < sizes[1]; j++) {
        j1 = j + nbdy;
        for (i = 0; i < nbdy; i++) {
            i1 = nbdy + sizes[0] + i;
            t2d[j1][i]  = bdry[0];
            t2d[j1][i1] = bdry[1];
        }
    }
}
else if (dim == 3) {
    t3d = m->t3d;
    for (k = 0; k < nbdy; k++) {
        k1 = k + sizes[2] + nbdy;
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                t3d[k][j][i]  = bdry[4];
                t3d[k1][j][i] = bdry[5];
            }
        }
    }
    for (k = 0; k < sizes[2]; k++) {
```

```
            k1 = nbdy + k;
            for (j = 0; j < nbdy; j++) {
                j1 = nbdy + sizes[1] + j;
                for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                    t3d[k1][j][i]  = bdry[2];
                    t3d[k1][j1][i] = bdry[3];
                }
            }
        }
        for (k = 0; k < sizes[2]; k++) {
            k1 = nbdy + k;
            for (j = 0; j < sizes[1]; j++) {
                j1 = nbdy + j;
                for (i = 0; i < nbdy; i++) {
                    i1 = nbdy + sizes[0] + i;
                    t3d[k1][j1][i]  = bdry[0];
                    t3d[k1][j1][i1] = bdry[1];
                }
            }
        }
    }
    return 0;
 }

int set_T_bdry1d(H_Mesh *m)
{
/*  set extended boundary conditions for checking the correctness */

    int d, i, i0, i1, i0_src, i1_src;
    int *sizes;
    double sloph0, slopn0, sloph1, slopn1;
    double dc[3], *coordl, *coordr, *th, *tn;

    if (dim != 1) return 0;
    if (m->parent) return 0;
```

```
    sizes = m->sizes;
    coordl = m->coordl;
    coordr = m->coordr;
    for (d = 0; d < dim; d++) {
        dc[d] = (coordr[d] - coordl[d])/(double) sizes[d];
    }
    i1_src = sizes[0] + nbdy - 1;
    th = m->th1d;
    tn = m->tn1d;
    sloph0 = 2.0 * (th[nbdy] - tbdry[0])/dc[0];
    slopn0 = 2.0 * (tn[nbdy] - tbdry[0])/dc[0];
    sloph1 = 2.0 * (tbdry[1] - th[i1_src])/dc[0];
    slopn1 = 2.0 * (tbdry[1] - tn[i1_src])/dc[0];

    for (i = 0; i < nbdy; i++) {
        th[i] = th[nbdy] - ((double)(nbdy - i)) * dc[0] * sloph0;
        tn[i] = tn[nbdy] - ((double)(nbdy - i)) * dc[0] * slopn0;
        i1 = nbdy + sizes[0] + i;
        th[i1] = th[i1_src] + ((double)(i+1)) * dc[0] * sloph1;
        tn[i1] = tn[i1_src] + ((double)(i+1)) * dc[0] * slopn1;
    }
    return 0;
}

int set_cgc_bdry1d(double *cgch, double *cgcn, int *sizes)
{
/*  set boundary condition for coarse grid correction in 1D */

    int i, i0, i1, i0_src, i1_src;

    for (i = 0; i < nbdy; i++) {
        i0_src = (nbdy + nbdy - i - 1);
        i1_src = sizes[0] + nbdy - 1 - i;
        i1 = sizes[0] + nbdy + i;
        cgch[i]  = - cgch[i0_src];
        cgch[i1] = - cgch[i1_src];
```

```
        cgcn[i]  = - cgcn[i0_src];
        cgcn[i1] = - cgcn[i1_src];
    }
    return 0;
 }

int set_cgc_bdry2d(double **cgch, double **cgcn, int *sizes)
{
/*  set boundary condition for coarse grid correction in 2D */

    int i, i0, i1, i0_src, i1_src, j, j0, j1, j0_src, j1_src;

    for (j = 0; j < nbdy; j++) {
        j0_src = (nbdy + nbdy - j - 1);
        j1_src = sizes[1] + nbdy - 1 - j;
        j1     = sizes[1] + nbdy + j;
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            cgch[j][i]  = - cgch[j0_src][i];
            cgch[j1][i] = - cgch[j1_src][i];

            cgcn[j][i]  = - cgcn[j0_src][i];
            cgcn[j1][i] = - cgcn[j1_src][i];
        }
    }
    for (j0 = 0; j0 < sizes[1]; j0++) {
        j = nbdy + j0;
        for (i = 0; i < nbdy; i++) {
            i0_src = (nbdy + nbdy - i - 1);
            i1_src = sizes[0] + nbdy - 1 - i;
            i1 = sizes[0] + nbdy + i;
            cgch[j][i]  = - cgch[j][i0_src];
            cgch[j][i1] = - cgch[j][i1_src];

            cgcn[j][i]  = - cgcn[j][i0_src];
            cgcn[j][i1] = - cgcn[j][i1_src];
```

```
        }
    }
    return 0;
 }

int set_cgc_bdry3d(double ***cgch, double ***cgcn, int *sizes)
{
/*  set boundary condition for coarse grid correction in 3D */

    int i, i0, i1, i0_src, i1_src, j, j0, j1, j0_src, j1_src;
    int k, k0, k1, k0_src, k1_src;

    for (k = 0; k < nbdy; k++) {
        k0_src = (nbdy + nbdy - k - 1);
        k1_src = sizes[2] + nbdy - 1 - k;
        k1     = sizes[2] + nbdy + k;
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                 cgch[k][j][i]  = - cgch[k0_src][j][i];
                 cgch[k1][j][i] = - cgch[k1_src][j][i];
                 cgcn[k][j][i]  = - cgcn[k0_src][j][i];
                 cgcn[k1][j][i] = - cgcn[k1_src][j][i];
            }
        }
    }
    for (k0 = 0; k0 < sizes[2]; k0++) {
        k = nbdy + k0;
        for (j = 0; j < nbdy; j++) {
            j0_src = (nbdy + nbdy - j - 1);
            j1_src = sizes[1] + nbdy - 1 - j;
            j1     = sizes[1] + nbdy + j;
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                cgch[k][j][i]  = - cgch[k][j0_src][i];
                cgch[k][j1][i] = - cgch[k][j1_src][i];

                cgcn[k][j][i]  = - cgcn[k][j0_src][i];
```

```
                cgcn[k][j1][i] = - cgcn[k][j1_src][i];
            }
        }
    }
    for (k0 = 0; k0 < sizes[2]; k0++) {
        k = nbdy + k0;
        for (j0 = 0; j0 < sizes[1]; j0++) {
            j = nbdy + j0;
            for (i = 0; i < nbdy; i++) {
                i0_src = (nbdy + nbdy - i - 1);
                i1_src = sizes[0] + nbdy - 1 - i;
                i1 = sizes[0] + nbdy + i;
                cgch[k][j][i]  = - cgch[k][j][i0_src];
                cgch[k][j][i1] = - cgch[k][j][i1_src];

                cgcn[k][j][i]  = - cgcn[k][j][i0_src];
                cgcn[k][j][i1] = - cgcn[k][j][i1_src];
            }
        }
    }
    return 0;
 }


int set_heat_coef(H_Mesh *m)
{
/*  set heat coefficients */

    int i, j, k, imat, d;
    int *sizes, offsets[3], msizes[3], sizes_ext[3];
    double *coordl, *coordr, *mcoordl, *mcoordr, coordl_ext[3];
    double coef, coef_max, dc, sml, dcinv[3], dc2inv[3];
    double *c1d, **c2d, ***c3d;
    double *cx1d, **cx2d, **cy2d, ***cx3d, ***cy3d, ***cz3d;

    c1d = m->c1d;
```

```
c2d = m->c2d;
c3d = m->c3d;

sizes  = m->sizes;
coordl = m->coordl;
coordr = m->coordr;
sml = 0.01 *(coordr[0] - coordl[0])/(double)sizes[0];
for (d = 0; d < dim; d++) {
    sizes_ext[d] = sizes[d] + nbdy + nbdy;
    dc = (coordr[d] - coordl[d])/(double)sizes[d];
    coordl_ext[d] = coordl[d] - dc * (double) nbdy;
    dcinv[d] = 1.0/dc;
    dc2inv[d] = dcinv[d] * dcinv[d];
}
if (dim == 1) {
    for (i = 0; i < sizes_ext[0]; i++) {
        c1d[i] = -1.0;
    }
}
else if (dim == 2) {
    for (j = 0; j < sizes_ext[1]; j++) {
        for (i = 0; i < sizes_ext[0]; i++) {
            c2d[j][i] = -1.0;
        }
    }
}
else if (dim == 3) {
    for (k = 0; k < sizes_ext[2]; k++) {
        for (j = 0; j < sizes_ext[1]; j++) {
            for (i = 0; i < sizes_ext[0]; i++) {
                c3d[k][j][i] = -1.0;
            }
        }
    }
}
coef_max = 0.0;
```

```
assert(nmat > 0);
for (imat = 0; imat < nmat; imat++) {
    coef = coef_array[imat];
    if (coef > coef_max) coef_max = coef;
    mcoordl = mat_coordl_array[imat];
    mcoordr = mat_coordr_array[imat];
    for (d = 0; d < dim; d++) {
        offsets[d] = (mcoordl[d] + sml - coordl_ext[d]) * dcinv[d];
        offsets[d] = MAX(offsets[d], 0);
        offsets[d] = MIN(offsets[d], sizes_ext[d]);
        msizes[d]  = (mcoordr[d] + sml - coordl_ext[d]) * dcinv[d];
        msizes[d]  = MAX(msizes[d], 0);
        msizes[d]  = MIN(msizes[d], sizes_ext[d]);
    }
    if (dim == 1) {
        for (i = offsets[0]; i < msizes[0]; i++) {
            c1d[i] = coef;
        }
    }
    else if (dim == 2) {
        for (j = offsets[1]; j < msizes[1]; j++) {
            for (i = offsets[0]; i < msizes[0]; i++) {
                c2d[j][i] = coef;
            }
        }
    }
    else if (dim == 3) {
        for (k = offsets[2]; k < msizes[2]; k++) {
            for (j = offsets[1]; j < msizes[1]; j++) {
                for (i = offsets[0]; i < msizes[0]; i++) {
                    c3d[k][j][i] = coef;
                }
            }
        }
    }
}
```

```
if (dim == 1) {
    for (i = 0; i < sizes_ext[0]; i++) {
        if (c1d[i] < 0.0) {
            err_msg("c1d[i] < 0.0");
        }
    }
}
else if (dim == 2) {
    for (j = 0; j < sizes_ext[1]; j++) {
        for (i = 0; i < sizes_ext[0]; i++) {
            if (c2d[j][i] < 0.0) {
                err_msg("c2d[j][i] < 0.0");
            }
        }
    }
}
else if (dim == 3) {
    for (k = 0; k < sizes_ext[2]; k++) {
        for (j = 0; j < sizes_ext[1]; j++) {
            for (i = 0; i < sizes_ext[0]; i++) {
                if (c3d[k][j][i] < 0.0) {
                   err_msg("c3d[k][j][i] < 0.0");
                }
            }
        }
    }
}
coefbydc2 = coef_max * dc2inv[0];

/* Set coefficients lambda_x, lambda_y, lambda_z */
if (dim == 1) {
    cx1d = m->cx1d;
    for (i = 1; i < sizes[0] + nbdy + nbdy; i++) {
        cx1d[i] = 2.0 *c1d[i-1] * c1d[i]/(c1d[i-1] + c1d[i]);
    }
}
```

```
else if (dim == 2) {
    cx2d = m->cx2d;
    cy2d = m->cy2d;
    for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
        for (i = 1; i < sizes[0] + nbdy + nbdy; i++) {
            cx2d[j][i] = 2.0 * c2d[j][i-1] * c2d[j][i]/(c2d[j][i-1] + c2d[j][i]);
        }
    }
    for (j = 1; j < sizes[1] + nbdy + nbdy; j++) {
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            cy2d[j][i] = 2.0 * c2d[j-1][i] * c2d[j][i]/(c2d[j-1][i] + c2d[j][i]);
        }
    }
}
else if (dim == 3) {
    cx3d = m->cx3d;
    cy3d = m->cy3d;
    cz3d = m->cz3d;
    for (k = 0; k < sizes[2] + nbdy + nbdy; k++) {
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 1; i < sizes[0] + nbdy + nbdy; i++) {
                cx3d[k][j][i] = 2.0 * c3d[k][j][i-1] * c3d[k][j][i]
                                    /(c3d[k][j][i-1] + c3d[k][j][i]);
            }
        }
    }
    for (k = 0; k < sizes[2] + nbdy + nbdy; k++) {
        for (j = 1; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                cy3d[k][j][i] = 2.0 * c3d[k][j-1][i] * c3d[k][j][i]
                                    /(c3d[k][j-1][i] + c3d[k][j][i]);
            }
        }
    }
    for (k = 1; k < sizes[2] + nbdy + nbdy; k++) {
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
```

```
                for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                    cz3d[k][j][i] = 2.0 * c3d[k-1][j][i] * c3d[k][j][i]
                                        /(c3d[k-1][j][i] + c3d[k][j][i]);
                }
            }
        }
    }
    return 0;
 }

int set_coef_fr_parent(H_Mesh *child)
{
/*  derive heat coefficients from parent */

    int i0, i1, i, ip0, ip1, ii, ip, j0, j1, j, jp0, jp1, jj, jp;
    int k0, k1, k, kp0, kp1, kk, kp, d;
    int *sizes, sizes_ext[3];
    double *coordl, *coordr, dc, dc2inv[3];
    double *c1d_parent, *c1d, *cx1d;
    double **c2d_parent, **c2d, **cx2d, **cy2d;
    double ***c3d_parent, ***c3d, ***cx3d, ***cy3d, ***cz3d;

    H_Mesh *parent;

    parent     = child->parent;
    c1d_parent = parent->c1d;
    c2d_parent = parent->c2d;
    c3d_parent = parent->c3d;

    c1d  = child->c1d;
    c2d  = child->c2d;
    c3d  = child->c3d;

    cx1d = child->cx1d;
    cx2d = child->cx2d;
    cy2d = child->cy2d;
```

```
cx3d = child->cx3d;
cy3d = child->cy3d;
cz3d = child->cz3d;

coordl = child->coordl;
coordr = child->coordr;
sizes  = child->sizes;
for (d = 0; d < dim; d++) {
    sizes_ext[d] = sizes[d] + nbdy + nbdy;
    dc = (coordr[d] - coordl[d])/(double)sizes[d];
    dc2inv[d] = 1.0/(dc * dc);
}
if (dim == 1) {
    for (i = 0; i < sizes[0]; i++) {
        c1d[i] = -1.0;
    }
    for (i0 = 0; i0 < sizes[0]; i0++) {   /** Interior cells **/
        i = nbdy + i0;
        ip0 = i0 + i0;
        ip1 = ip0 + 2;
        c1d[i] = 0;
        for (ii = ip0; ii < ip1; ii++) {
            ip = ii + nbdy;
            c1d[i] += c1d_parent[ip];
        }
        c1d[i] *= 0.5;
    }
    for (i = 0; i < nbdy; i++) {  /** boundary cells */
        i1  = nbdy + sizes[0] + i;
        ip0 = nbdy - 1;
        ip1 = sizes[0] + sizes[0] + nbdy;
        c1d[i]  = c1d_parent[ip0];
        c1d[i1] = c1d_parent[ip1];
    }
    for (i = 0; i < sizes[0]; i++) {
        if (c1d[i] < 0.0) {
```

```
            err_msg("c1d[i] < 0.0 in set_coef_fr_parent");
            exit(0);
        }
    }
    /*  lambdax  */
    for (i = 1; i < sizes[0] + nbdy + nbdy; i++) {
        cx1d[i] = 0.5 *c1d[i-1] * c1d[i]/(c1d[i-1] + c1d[i]);
    }
}
else if (dim == 2) {
    for (j = 0; j < sizes_ext[1]; j++) {
        for (i = 0; i < sizes_ext[0]; i++) {
            c2d[j][i] = -1.0;
        }
    }
    for (j0 = 0; j0 < sizes[1]; j0++) {
        j = nbdy + j0;
        jp0 = j0 + j0;
        jp1 = jp0 + 2;
        for (i0 = 0; i0 < sizes[0]; i0++) {
            i = nbdy + i0;
            ip0 = i0 + i0;
            ip1 = ip0 + 2;
            c2d[j][i] = 0;
            for (jj = jp0; jj < jp1; jj++) {
                jp = jj + nbdy;
                for (ii = ip0; ii < ip1; ii++) {
                    ip = ii + nbdy;
                    c2d[j][i] += c2d_parent[jp][ip];
                }
            }
            c2d[j][i] *= 0.25;
        }
    }
    for (j = 0; j < nbdy; j++) {    /* top and bottom bdry at j direction */
        j1 = nbdy + sizes[1] + j;
```

```
    jp0 = nbdy - 1;
    jp1 = nbdy + sizes[1] + sizes[1];
    for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
        if (i < nbdy) {
            ip = nbdy - 1;
        }
        else if (i < sizes[0] + nbdy) {
            ip = 2 * (i - nbdy) + nbdy;
        }
        else {
            ip = sizes[0] + sizes[0] + nbdy;
        }
        c2d[j][i]  = c2d_parent[jp0][ip];
        c2d[j1][i] = c2d_parent[jp1][ip];
    }
}
for (j = nbdy; j < sizes[1] + nbdy; j++) {  /* left and right bdry at i */
    jp = 2 * (j - nbdy) + nbdy;
    for (i = 0; i < nbdy; i++) {
        i1 = nbdy + sizes[0] + i;
        ip0 = nbdy - 1;
        ip1 = nbdy + sizes[0] + sizes[0];
        c2d[j][i]  = c2d_parent[jp][ip0];
        c2d[j][i1] = c2d_parent[jp][ip1];
    }
}
for (j = 0; j < sizes_ext[1]; j++) {
    for (i = 0; i < sizes_ext[0]; i++) {
        if (c2d[j][i] < 0.0) {
            err_msg("c2d[j][i] < 0.0 in set_coef_fr_parent");
            exit(0);
        }
    }
}
/*  lambdax, lambday */
for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
```

```
        for (i = 1; i < sizes[0] + nbdy + nbdy; i++) {
            cx2d[j][i] = 2.0 * c2d[j][i-1] * c2d[j][i]/(c2d[j][i-1] + c2d[j][i]);
        }
    }
    for (j = 1; j < sizes[1] + nbdy + nbdy; j++) {
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            cy2d[j][i] = 2.0 * c2d[j-1][i] * c2d[j][i]/(c2d[j-1][i] + c2d[j][i]);
        }
    }
}
else if (dim == 3) {
    for (k = 0; k < sizes_ext[2]; k++) {
        for (j = 0; j < sizes_ext[1]; j++) {
            for (i = 0; i < sizes_ext[0]; i++) {
                c3d[k][j][i] = -1.0;
            }
        }
    }
    for (k0 = 0; k0 < sizes[2]; k0++) {  /* interior cells */
        k = nbdy + k0;
        kp0 = k0 + k0;
        kp1 = kp0 + 2;
        for (j0 = 0; j0 < sizes[1]; j0++) {
            j = nbdy + j0;
            jp0 = j0 + j0;
            jp1 = jp0 + 2;
            for (i0 = 0; i0 < sizes[0]; i0++) {
                i = nbdy + i0;
                ip0 = i0 + i0;
                ip1 = ip0 + 2;
                c3d[k][j][i] = 0.0;
                for (kk = kp0; kk < kp1; kk++) {
                    kp = nbdy + kk;
                    for (jj = jp0; jj < jp1; jj++) {
                        jp = jj + nbdy;
                        for (ii = ip0; ii < ip1; ii++) {
```

```
                        ip = nbdy + ii;
                        c3d[k][j][i] += c3d_parent[kp][jp][ip];
                    }
                }
            }
            c3d[k][j][i] *= 0.125;
        }
    }
}
for (k = 0; k < nbdy; k++) {  /* top and bottom bdry at k direction */
    k1 = nbdy + sizes[2] + k;
    kp0 = nbdy - 1;
    kp1 = nbdy + sizes[2] + sizes[2];
    for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
        if (j < nbdy) {
            jp = nbdy - 1;
        }
        else if (j < sizes[1] + nbdy) {
            jp = 2 * (j - nbdy) + nbdy;
        }
        else {
            jp = sizes[1] + sizes[1] + nbdy;
        }
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            if (i < nbdy) {
                ip = nbdy - 1;
            }
            else if (i < sizes[0] + nbdy) {
                ip = 2 * (i - nbdy) + nbdy;
            }
            else {
                ip = sizes[0] + sizes[0] + nbdy;
            }
            c3d[k][j][i]  = c3d_parent[kp0][jp][ip];
            c3d[k1][j][i] = c3d_parent[kp1][jp][ip];
        }
```

```
    }
}
for (k0 = 0; k0 < sizes[2]; k0++) {  /* front and rear bdry at j direction */
    k  = nbdy + k0;
    kp = nbdy + k0 + k0;
    for (j = 0; j < nbdy; j++) {
        j1 = nbdy + sizes[1] + j;
        jp0 = nbdy - 1;
        jp1 = nbdy + sizes[1] + sizes[1];
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            if (i < nbdy) {
                ip = nbdy - 1;
            }
            else if (i < sizes[0] + nbdy) {
                ip = 2 * (i - nbdy) + nbdy;
            }
            else {
                ip = sizes[0] + sizes[0] + nbdy;
            }
            c3d[k][j][i]  = c3d_parent[kp][jp0][ip];
            c3d[k][j1][i] = c3d_parent[kp][jp1][ip];
        }
    }
}
for (k0 = 0; k0 < sizes[2]; k0++) {  /* left and right bdry at i direction */
    k  = nbdy + k0;
    kp = nbdy + k0 + k0;
    for (j0 = 0; j0 < sizes[1]; j0++) {
        j = nbdy + j0;
        jp = nbdy + j0 + j0;
        for (i = 0; i < nbdy; i++) {
            i1  = nbdy + sizes[0] + i;
            ip0 = nbdy - 1;
            ip1 = nbdy + sizes[0] + sizes[0];
            c3d[k][j][i]  = c3d_parent[kp][jp][ip0];
            c3d[k][j][i1] = c3d_parent[kp][jp][ip1];
```

```
        }
    }
}
for (k = 0; k < sizes_ext[2]; k++) {
    for (j = 0; j < sizes_ext[1]; j++) {
        for (i = 0; i < sizes_ext[0]; i++) {
            if (c3d[k][j][i] < 0.0) {
               err_msg("c3d[k][j][i] < 0.0 in set_coef_fr_parent");
               exit(0);
            }
        }
    }
}
/*  lambdax, lambday, lambdaz */
for (k = 0; k < sizes[2] + nbdy + nbdy; k++) {
    for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
        for (i = 1; i < sizes[0] + nbdy + nbdy; i++) {
            cx3d[k][j][i] = 2.0 * c3d[k][j][i-1] * c3d[k][j][i]
                                /(c3d[k][j][i-1] + c3d[k][j][i]);
        }
    }
}
for (k = 0; k < sizes[2] + nbdy + nbdy; k++) {
    for (j = 1; j < sizes[1] + nbdy + nbdy; j++) {
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            cy3d[k][j][i] = 2.0 * c3d[k][j-1][i] * c3d[k][j][i]
                                /(c3d[k][j-1][i] + c3d[k][j][i]);
        }
    }
}
for (k = 1; k < sizes[2] + nbdy + nbdy; k++) {
    for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            cz3d[k][j][i] = 2.0 * c3d[k-1][j][i] * c3d[k][j][i]
                                /(c3d[k-1][j][i] + c3d[k][j][i]);
        }
```

```
            }
        }
    }
    return 0;
 }
int set_gause_1d(H_Mesh *m)
{
/*  set gause profile for an example of convergence */

    int i0, i, d;
    int *sizes;
    double width, hl, tmp, t_add, *coordl, *coordr, ctr[3], dc[3], x;
    double *t1d;

    if (dim != 1) return 0;
    if (m->parent) return 0;

    sizes  = m->sizes;
    coordl = m->coordl;
    coordr = m->coordr;

    if (tbdry[0] != tbdry[1]) {
        printf("ERROR: tbdry[0] != tbdry[1]\n");
        exit(0);
    }
    width = 0.25 * (coordr[0] - coordl[0]);
    hl    = 0.5 * (coordr[0] - coordl[0]);
    tmp = hl/width;
    t_add  = tbdry[0] - t_initial * exp(-tmp * tmp);
    for (d = 0; d < dim; d++) {
        ctr[d] = 0.5 *(coordr[d] - coordl[d]);
        dc[d] = (coordr[d] - coordl[d])/(double)sizes[d];
    }
    t1d = m->t1d;
    x = coordl[0] - 0.5 * dc[0];
    for (i0 = 0; i0 < sizes[0]; i0++) {
```

```
        i = nbdy + i0;
        x += dc[0];
        tmp = (x - ctr[0])/width;
        t1d[i] = t_add + t_initial * exp(-tmp * tmp);
    }
    return 0;
}


int set_initial_temp(H_Mesh *m)
{
    int i, j, k;
    int *sizes;
    double *t1d, **t2d, ***t3d;
    double *sh1d, **sh2d, ***sh3d, *sn1d, **sn2d, ***sn3d;
    H_Mesh *child;

    sizes = m->sizes;
    t1d = m->t1d;
    t2d = m->t2d;
    t3d = m->t3d;
    sh1d = m->sh1d;
    sn1d = m->sn1d;
    sh2d = m->sh2d;
    sn2d = m->sn2d;
    sh3d = m->sh3d;
    sn3d = m->sn3d;

    if (dim == 1) {
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            t1d[i] = t_initial;
            sh1d[i] = 0.0;
            sn1d[i] = 0.0;
        }
    }
    else if (dim == 2) {
```

```
    for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            t2d[j][i] = t_initial;
            sh2d[j][i] = 0.0;
            sn2d[j][i] = 0.0;
        }
    }
}
else if (dim == 3) {
    for (k = 0; k < sizes[2] + nbdy + nbdy; k++) {
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                t3d[k][j][i] = t_initial;
                sh3d[k][j][i] = 0.0;
                sn3d[k][j][i] = 0.0;
            }
        }
    }
}
child = m->child;
while (child) {
     sizes = m->sizes;
     t1d = child->t1d;
     t2d = child->t2d;
     t3d = child->t3d;

     if (dim == 1) {
         for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
             t1d[i] = 0.0;
         }
     }
     else if (dim == 2) {
         for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
             for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                 t2d[j][i] = 0.0;
             }
```

```
            }
        }
        else if (dim == 3) {
            for (k = 0; k < sizes[2] + nbdy + nbdy; k++) {
                for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
                    for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                        t3d[k][j][i] = 0.0;
                    }
                }
            }
        }
    }
    return 0;
}

int set_mesh(double *coordl, double *coordr, int *sizes, H_Mesh *m)
{
    double **t2d, **tn2d, **th2d, **c2d, **cx2d, **cy2d;
    double ***t3d, ***tn3d, ***th3d, ***c3d, ***cx3d, ***cy3d, ***cz3d;
    double **rn2d, **rh2d, ***rn3d, ***rh3d;
    double **sn2d, **sh2d, ***sn3d, ***sh3d;
    int    lsize, d;

    assert(coordl);
    assert(coordr);
    assert(sizes);
    assert(m);

    init_mesh(m);
    m->level = 0;
    memcpy(m->sizes,  sizes,  (size_t)(dim * sizeof(int)));
    memcpy(m->coordl, coordl, (size_t)(dim * sizeof(double)));
    memcpy(m->coordr, coordr, (size_t)(dim * sizeof(double)));

    if (dim == 1) {
       lsize   = sizes[0] + nbdy + nbdy;
```

```
    m->t1d  = (double *) malloc(lsize * sizeof(double));
    m->tn1d = (double *) malloc(lsize * sizeof(double));
    m->th1d = (double *) malloc(lsize * sizeof(double));
    m->tn1d = (double *) malloc(lsize * sizeof(double));
    m->th1d = (double *) malloc(lsize * sizeof(double));
    m->sn1d = (double *) malloc(lsize * sizeof(double));
    m->sh1d = (double *) malloc(lsize * sizeof(double));
    m->rn1d = (double *) malloc(lsize * sizeof(double));
    m->rh1d = (double *) malloc(lsize * sizeof(double));
    m->c1d  = (double *) malloc(lsize * sizeof(double));
    m->cx1d = (double *) malloc(lsize * sizeof(double));
}
else if (dim == 2) {
    set_2d_form(sizes, nbdy, &t2d);
    set_2d_form(sizes, nbdy, &th2d);
    set_2d_form(sizes, nbdy, &tn2d);
    set_2d_form(sizes, nbdy, &sh2d);
    set_2d_form(sizes, nbdy, &sn2d);
    set_2d_form(sizes, nbdy, &rh2d);
    set_2d_form(sizes, nbdy, &rn2d);
    set_2d_form(sizes, nbdy, &c2d);
    set_2d_form(sizes, nbdy, &cx2d);
    set_2d_form(sizes, nbdy, &cy2d);

    m->t2d  = t2d;
    m->th2d = th2d;
    m->tn2d = tn2d;
    m->sh2d = sh2d;
    m->sn2d = sn2d;
    m->rh2d = rh2d;
    m->rn2d = rn2d;
    m->c2d  = c2d;
    m->cx2d  = cx2d;
    m->cy2d  = cy2d;
}
else if (dim == 3) {
```

```
        set_3d_form(sizes, nbdy, &t3d);
        set_3d_form(sizes, nbdy, &th3d);
        set_3d_form(sizes, nbdy, &tn3d);
        set_3d_form(sizes, nbdy, &sh3d);
        set_3d_form(sizes, nbdy, &sn3d);
        set_3d_form(sizes, nbdy, &rh3d);
        set_3d_form(sizes, nbdy, &rn3d);
        set_3d_form(sizes, nbdy, &c3d);
        set_3d_form(sizes, nbdy, &cx3d);
        set_3d_form(sizes, nbdy, &cy3d);
        set_3d_form(sizes, nbdy, &cz3d);

        m->t3d  = t3d;
        m->th3d = th3d;
        m->tn3d = tn3d;
        m->sh3d = sh3d;
        m->sn3d = sn3d;
        m->rh3d = rh3d;
        m->rn3d = rn3d;
        m->c3d  = c3d;
        m->cx3d  = cx3d;
        m->cy3d  = cy3d;
        m->cz3d  = cz3d;
    }

    return 0;
}

int init_mesh(H_Mesh *m)
{
/*  Initialize mesh to NULL */

    int d;
    assert(m);
    m->level = -1;
    for (d = 0; d < dim; d++) {
```

```
        m->sizes[d] = 0;
        m->coordl[d] = 0.0;
        m->coordr[d] = 0.0;
        m->t1d = NULL;
        m->t2d = NULL;
        m->t3d = NULL;
        m->c1d = NULL;
        m->c1d = NULL;
        m->c1d = NULL;
    }
    m->parent = NULL;
    m->child  = NULL;
    return 0;
}

int set_2d_form(int *sizes, int nbdy, double ***data2d)
{
/*  set 1D array to a form of 2D */

    int d, j;
    int lsize, sizes_ext[3];

    assert(sizes);

    lsize = 1;
    for (d = 0; d < dim; d++) {
        sizes_ext[d] = sizes[0] + nbdy + nbdy;
        lsize *= sizes_ext[d];
    }
    *data2d = (double **) malloc(sizes_ext[1] * sizeof(double *));
    (*data2d)[0] = (double *) malloc(lsize * sizeof(double));
    for (j = 1; j < sizes_ext[1]; j++) {
        (*data2d)[j] = (*data2d)[j-1] + sizes_ext[0];
    }
    return 0;
 }
```

```
int set_3d_form(int *sizes, int nbdy, double ****data3d)
{
/*  set 1D array to a form of 3D */

    int d, k, j, offset1, offset2;
    int lsize, sizes_ext[3];
    double *r1d;

    assert(sizes);

    lsize = 1;
    for (d = 0; d < dim; d++) {
        sizes_ext[d] = sizes[0] + nbdy + nbdy;
        lsize *= sizes_ext[d];
    }
    r1d = (double *)malloc(lsize * sizeof(double));
    *data3d = (double ***) malloc(sizes_ext[2] * sizeof(double **));
    (*data3d)[0] = (double **) malloc(sizes_ext[1] * sizes_ext[2] * sizeof(double *));
    offset1 = 0;
    offset2 = 0;
    for (k = 0; k < sizes_ext[2]; k++) {
        (*data3d)[k] = (*data3d)[0] + offset2;
        for (j = 0; j < sizes_ext[1]; j++) {
            (*data3d)[k][j] = r1d + offset1;
            offset1 += sizes_ext[0];
        }
        offset2 += sizes_ext[1];
     }
     return 0;
 }

int dump(H_Mesh *m, double t, int idump)
{
/*  write temperature and coefficient to a file */
```

```
char name[125];
char mesh_name[] = "mesh";
char var_name[]  = "T";
char coef_name[] = "coef";
int fileid, d, i0, i, ii, j0, j, k0, k;
int lsize, *sizes;
double dc[3], x, y, z, *coordl, *coordr;
double *v, *c, *c1d, **c2d, ***c3d, *t1d, **t2d, ***t3d;
mio_Structured_Mesh mesh;
mio_Mesh_Var_Type type;
mio_Mesh_Var var;

FILE *fp;

c1d   = m->c1d;
c2d   = m->c2d;
c3d   = m->c3d;

if (t == 0.0) {
   t1d = m->t1d;
   t2d = m->t2d;
   t3d = m->t3d;
}
else {
   t1d = m->tn1d;
   t2d = m->tn2d;
   t3d = m->tn3d;
}
sizes = m->sizes;
coordl = m->coordl;
coordr = m->coordr;
for (d = 0; d < dim; d++) {
    dc[d] = (coordr[d] - coordl[d])/(double) sizes[d];
}
if (dim == 1) {
    sprintf(name, "data_%05d.txt", idump);
```

```
    printf("write %s ...\n", name);
    fp = fopen(name, "w+");
    fprintf(fp, "# t = %e\n", t);
    fprintf(fp, "# nmat  %d,  coef_array = ", nmat);
    for (i = 0; i < nmat; i++) {
        fprintf(fp, "%e  ", coef_array[i]);
    }
    fprintf(fp, "\n");

    fprintf(fp, "# T(%e) = %e, T(%e) = %e\n",
            coordl[0]-0.5*dc[0], tbdry[0], coordr[0] + 0.5*dc[0], tbdry[1]);
    fprintf(fp, "# courant = %e\n", courant);
    fprintf(fp, "    i         x          T             coef\n");
    x = coordl[0] - 0.5 * dc[0];
    for (i = nbdy; i < sizes[0] + nbdy; i++) {
        x += dc[0];
        fprintf(fp, "%5d  %e  %e  %e\n", i, x, t1d[i], c1d[i]);
    }
    fclose(fp);
}
else {
    sprintf(name, "data_%05d", idump);
    mio_open_file(name, mio_file_create, &fileid);
    mio_init(mio_smesh, -1, &mesh);

    mesh.name = mesh_name;
    mesh.dims = dim;
    mesh.datatype = mio_double;
    mesh.element_centered = 1;
    lsize = 1.0;
    for (d = 0; d < dim; d++) {
        mesh.coordmin[d] = coordl[d];
        mesh.dcoord[d] = dc[d];
        mesh.sizes[d]  = sizes[d];
        lsize *= sizes[d];
    }
```

```
mio_write(mio_smesh, fileid, &mesh);


v = (double *) malloc((lsize + lsize) * sizeof(double));
c = v + lsize;
if (dim == 2) {
    ii = 0;
    for (j0 = 0; j0 < sizes[1]; j0++) {
        j = nbdy + j0;
        for (i0 = 0; i0 < sizes[0]; i0++) {
            i = nbdy + i0;
            v[ii] = t2d[j][i];
            c[ii] = c2d[j][i];
            ii++;
        }
    }
    type = mio_face;
}
else if (dim == 3) {
    ii = 0;
    for (k0 = 0; k0 < sizes[2]; k0++) {
        k = nbdy + k0;
        for (j0 = 0; j0 < sizes[1]; j0++) {
            j = nbdy + j0;
            for (i0 = 0; i0 < sizes[0]; i0++) {
                i = nbdy + i0;
                v[ii] = t3d[k][j][i];
                c[ii] = c3d[k][j][i];
                ii++;
            }
        }
    }
    type = mio_zone;
}
mio_init(mio_mesh_var, -1, &var);
var.name = var_name;
var.mesh_ids[0] = mesh.id;
```

```
        var.num_meshes = 1;
        var.type = type;
        var.rank = 0;
        var.datatype = mio_double;
        var.comps[0].buffer = v;
        mio_write(mio_mesh_var, fileid, &var);

        mio_init(mio_mesh_var, -1, &var);
        var.name = coef_name;
        var.mesh_ids[0] = mesh.id;
        var.num_meshes = 1;
        var.type = type;
        var.rank = 0;
        var.datatype = mio_double;
        var.comps[0].buffer = c;
        mio_write(mio_mesh_var, fileid, &var);

        mio_close_file(fileid);

        free(v);
    }
    return 0;
}

void err_msg(char *msg)
{
    printf("ERROR: %s\n", msg);
    abort();
    return;
}
```