Simulations for Radiation Diffusion in Nuclear Accidents

New Mexico

Supercomputing Challenge

Final Report

April 5, 2010

Team Number 66
Los Alamos High School

Team Members
Edward Dai
Judy Lee

Teacher: Lee Goodwin
Mentor: William Dai

Executive Summary

Developing a computer model, numerical methods, and computer codes to simulate the spread of radiation energy in the case of numerical accidents are the tasks of this project. There are certain difficulties in the development of numerical methods involved in this project. Among them are accuracy of methods, discontinuity of radiative conductivity between materials involved, convergence of iterations, and the nonlinearity of the basic equation. Some of these issues are resolved in this project, but others remain to be better addressed in the future.

Some numerical methods are first order accurate, and others are second or higher order accurate, i.e., more accurate than the first order methods. Some methods work only for time-dependent problems, but others work only for steady states. When a system has two or more materials involved, the diffusion property of material experiences a discontinuous change across the interfaces between materials. This discontinuity introduces a fundamental difficulty for numerical methods, since most methods are based on mathematical approximations, for example, Taylor expansions, which are invalid across any discontinuity. When the discontinuity is strong, these mathematical approximations will results in obvious numerical errors.

We have developed a numerical method and computer codes for both time-dependent (transient) problems and steady states of one-, two-, and three-dimensional diffusion equations in systems with different materials. The focus of the development is on the capability for both transient problems and steady states, second order accuracy, iterative approach, and the discontinuity of material properties. The iterative approach is typically more efficient to solve large systems of algebraic equations than its counterpart, direct approach. Our method is second order accurate in both space and time. The discontinuity between different materials is correctly treated based on the governing physics principle. An iterative approach is proposed, and the multigrid method is applied, which is an advance technique to speed up the convergence of iterative approaches.

The issue of the nonlinearity in the treatment of material interfaces has been appropriately addressed, but iterative solvers for the system of fully nonlinear equations remain to be developed. Currently we use the values of unknowns at the previous time step in the evaluation of radiative conductivity. Although we intent to simulate the diffusion of a real nuclear accident, we are unable to finish the implementation for a real accident. Instead we simulated an artificial one.

1. Statement of the Problem

In physics, radiation describes a process in which energetic particles or waves travel through a medium or space. There are two distinct types of radiation; ionizing and non-ionizing. The word radiation is commonly used in reference to ionizing radiation only (i.e., having sufficient energy to ionize an atom), but it may also refer to non-ionizing radiation (e.g., radio waves or visible light). The energy radiates (i.e., travels outward in straight lines in all directions) from its source. Both ionizing and non-ionizing radiation can be harmful to human and can result in changes to the natural environment.

A diffusion approximation provides a reasonably accurate description of penetration of radiation from a hot source to a cold medium. This approximation features a nonlinear process that leads to formation of a sharp thermal front, in which the solution can vary several orders of magnitude over a very short distance. Furthermore, different materials involved may have dramatically different conduction properties. Solutions of radiation diffusion may be discontinuous over space. Correct treatment for the discontinuity in the system may be challenging.

Methods to solve the radiation diffusion equation may be divided into explicit and implicit ones. 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. In the system of dramatically different materials, the material of the largest radiative conductivity dictates the time step. Therefore, an explicit method is inefficient for the systems with very different materials. 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. For a given time step, the solution in the material of large radiative conductivity may have reached a constant, which it varies in other materials. Therefore, the methods to be used must be able to give the correct steady state when the time step is very large. Of course, implicit methods normally involve solving a large set of algebraic equations at each time step.

Correct treatment for dramatically different materials is also a challenge. Typically, mathematical approximations are used for the interfaces between different materials, which will cause large numerical errors when materials involved are dramatically different.

In this project we will develop a numerical method for the radiation diffusion equation, which is second order accurate in both space and time. The method works for both explicit and implicit regions, and therefore after each time step the solution in some materials are almost constant, while in others it varies. When a time step is very large, the method gives the correct steady state. The different materials are treated based on physics laws across the interface between two materials instead of mathematical approximations, and therefore, there is no mathematical errors introduced because of the discontinuity of the material property. We plan to solve the nonlinearity together with the implicitness of system through iteration, and therefore there is not additional level of iterations needed for the nonlinearity, which significantly reduces the computational cost, but we have only implemented a simpler but less advanced approach for the nonlinearity.

2. Nuclear Accidents

A nuclear and radiation accident is defined by the International Atomic Energy Agency as "an event that has led to significant consequences to people, the environment or the facility. Examples include lethal effects to individuals, large radioactivity release to the environment, or reactor core melt."[8] The prime example of a "major nuclear accident" is one in which a reactor core is damaged and large amounts of radiation are released, such as in the Chernobyl Disaster in 1986 [9], as shown in Fig.1 and the nuclear accident following an earthquake, tsunami, and multiple fires and hydrogen explosions at Fukushima nuclear power plant of Japan on March 2, 2011 [10], as shown in Fig.2.

The likelihood and potential impact of nuclear accidents has been a topic of debate practically since the first nuclear reactors were constructed. It has also been a key factor in public concern about nuclear facilities. Many technical measures to reduce the risk of accidents or to minimize the amount of radioactivity released to the environment have been adopted.

It has been reported that worldwide there have been 100 accidents at nuclear power plants [11]. Fifty-eight accidents have occurred since the Chernobyl disaster, and almost two-thirds (56 out of 100) of all nuclear-related accidents have occurred in the United States. Serious radiation accidents include the radiotherapy accident in Costa Rica in 1996, radiotherapy accident in Zaragoza of Spain in 1990, radiation accident in Morocco in 1984, Goiania accident of Brazil in 1987, radiation accident in Mexico City in 1962, the Mayapuri radiological accident of 2010 in India, Chernobyl Disaster in 1986, and the accident on March 12, 2011 at Fukushima nuclear power plant of Japan.

Figure 1. The nuclear reactor after the disaster April 26, 1986, Pripyat, Ukrainian SSR, Soviet Union. Reactor 4 (center). Turbine building (lower left). Reactor 3 (center right).

Figure 2. The second explosion at Janpan nuclear plant: hydrogen blast occurs at Fukushima nuke plant's No.3 reactor on March 13, 2011.

One kind of nuclear accidents involves off site release of both radiation (gamma and neutron) and a radioactivity, although the release are often very limited. Very serious accidents may involve the release of radioactivity into the environment.

This project will numerically study how nuclear radiation energy will diffuse to limited off site or environment if shielding of radiation were damaged.

3. Diffusion Equation and Numerical Methods

Propagation of a radiation field and its interaction with materials can be modeled by an integro-differential equation that accounts for transport, emission, absorption, and scattering of photons. When the radiation field is isotropic, detailed treatment of transport in angle is not needed. The dependence on angle can be averaged out to obtain a description based on spectral energy density and flux. In a static medium at local thermal equilibrium, absorption is independent of frequency, so dependence on frequency can also be averaged out. In this case, the radiative flux is proportional to the gradient of the energy density, with the radiative conductivity proportional to the inverse of the opacity of the medium. Also, when the medium is in thermal equilibrium, emission equals absorption, and the total radiation energy density E is proportional to T^4, where T is the temperature of the medium. These considerations lead to a parabolic partial differential equation [13].

$$\frac{\partial E}{\partial t} = \nabla \cdot [\lambda(E) \nabla E] + s,$$

with initial conditions $E(t=0) = E_0(x, y, z)$. Here s is the source. We use a model of radiative conductivity in which $\lambda(E)$ may be a function of energy [14].

We write the basic equation in the form

$$\frac{\partial E}{\partial t} + \nabla \cdot J = s. \qquad (1)$$

Here J is the flux defined as

$$J \equiv -\nabla(\lambda E). \qquad (2)$$

Here the radiative conductivity $\lambda(E)$ may be very different for different materials.

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

$$E_{i,j,k}^n = E_{i,j,k} + \frac{\Delta t}{\Delta x}(\bar{J}_{xi,j,k} - \bar{J}_{xi+1,j,k}) + \frac{\Delta t}{\Delta y}(\bar{J}_{yi,j,k} - \bar{J}_{yi,j+1,k}) + \frac{\Delta t}{\Delta z}(\bar{J}_{zi,j,k} - \bar{J}_{zi,j,k+1}) + \bar{s}\Delta t. \qquad (3)$$

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

$$E_{i,j,k}^n \equiv \frac{1}{\Delta x \Delta y \Delta z} \int_{\Delta V} E(\Delta t, x, y, z) dxdydz,$$

$$\bar{J}_{xi,j,k} \equiv \frac{1}{\Delta t \Delta y \Delta z} \int_0^{\Delta t} \int_{y_j}^{y_{j+1}} \int_{z_k}^{z_{k+1}} J_x(t, x_i, y, z) dydzdt. \qquad (4)$$

In Eq.(3), the superscript n stands for the "new" time $t = \Delta t$. If we know the way to calculate fluxes needed in Eq.(3), we can find the temperature at $t = \Delta t$ from the given initial condition and source function. Therefore, one of the major tasks in numerical methods is to approximately find the fluxes.

If the time-averaged fluxes are replaced by their initial values, the approximation results in Euler forward method. If the fluxes are replaced by the values at the end of the time step, the scheme is called Euler backward scheme. Both Euler forward and backward schemes are first order accurate. One of the good features of Euler backward scheme is that numerical errors undergo quick damping for large time steps, and this feature is very useful for steady state problems. If the time-average fluxes are replaced by their averaged values at $t = 0$ and $t = \Delta t$, the result is the Crank-Nicolson scheme, which is second order accurate in time. But the numerical errors in Crank-Nicolson scheme do not damp out for large time steps, and therefore the scheme is not good for steady states.

To introduce quick damping for numerical errors within the second order accuracy, we introduce an additional time step level, as demonstrated in the scheme for hydrodynamics [5]. Within the second order accuracy, we approximately evaluate the time-averaged flux at $t = \Delta t/2$, and Eq.(3) becomes

$$E_{i,j,k}^n = E_{i,j,k} + \frac{\Delta t}{\Delta x}(J_{xi,j,k}^h - J_{xi+1,j,k}^h) + \frac{\Delta t}{\Delta y}(J_{yi,j,k}^h - J_{yi,j+1,k}^h) + \frac{\Delta t}{\Delta z}(J_{zi,j,k}^h - J_{zi,j,k+1}^h) + s^h \Delta t. \qquad (5)$$

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 as, for example,

$$J^h_{xi,j,k} \equiv \frac{1}{\Delta y \Delta z} \int_{z_k}^{z_{k+1}} \int_{y_j}^{y_{j+1}} J_x(\frac{\Delta t}{2}, x_{i,} y, z) dydz, \qquad (6)$$

As shown in Eq.(3), the equations to determine the temperature at the half time step may be similarly obtained,

$$E^h_{i,j,k} = E_{i,j,k} + \frac{\Delta t}{2\Delta x}(\bar{J}^h_{xi,j,k} - \bar{J}^h_{xi+1,j,k}) + \frac{\Delta t}{2\Delta y}(\bar{J}^h_{yi,j,k} - \bar{J}^h_{yi,j+1,k}) + \frac{\Delta t}{2\Delta z}(\bar{J}^h_{zi,j,k} - \bar{J}^h_{zi,j,k+1}) + \frac{1}{2}\bar{s}^h \Delta t. \qquad (7)$$

Here $E^h_{i,j,k}$ and fluxes, for example, $\bar{J}^h_{xi,j,k}$, are defined as

$$E^h_{i,j,k} \equiv \frac{1}{\Delta x \Delta y \Delta z} \int_{\Delta V} E(\frac{\Delta t}{2}, x, y, z) dxdydz, \qquad (8)$$

$$\bar{J}^h_{xi,j,k} \equiv \frac{2}{\Delta t \Delta y \Delta z} \int_0^{\Delta t/2} \int_{y_j}^{y_{j+1}} \int_{z_k}^{z_{k+1}} J_x(t, x_{i,} y, z) dydzdt, \qquad (9)$$

The time-averaged fluxes involved in Eq.(7) 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 temperature when a time step is very large. Therefore, our interpolation for the time-averaged fluxes is uniquely determined by values at $t = \Delta t / 2$ and $t = \Delta t$. Thus, the time-averaged fluxes in Eq.(7) are approximately obtained through the following approximation:

$$\bar{J}^h_{xi,j,k} \approx \frac{3}{2} J^h_{xi,j,k} - \frac{1}{2} J^n_{xi,j,k}, \quad \bar{J}^h_{yi,j,k} \approx \frac{3}{2} J^h_{yi,j,k} - \frac{1}{2} J^n_{yi,j,k}, \quad \bar{J}^h_{zi,j,k} \approx \frac{3}{2} J^h_{zi,j,k} - \frac{1}{2} J^n_{zi,j,k}. \qquad (10)$$

Here $J^n_{xi,j,k}$, $J^n_{yi,j,k}$, and $J^n_{zi,j,k}$ have the same definitions as Eq.(6) 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 $t = \Delta t / 2$.

The second open issue we would like to address is the numerical treatment of the discontinuity of the radiative conductivity. If the material in the cell (i,j,k) is the same as those in its neighboring cells and that the radiative conductivity is independent of E, the central difference may be used to approximately calculate the flux at cell interface, for example,

$$J^h_{xi,j,k} \approx -\frac{\lambda}{\Delta x}(E^h_{i,j,k} - E^h_{i-1,j,k}). \quad (11)$$

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.(11) is no longer true across material interfaces. A typical previous method is to use the following,

$$\lambda \approx \frac{1}{2}(\lambda_{i-1,j,k} + \lambda_{i,j,k}). \quad (11b)$$

But, we claim that this formula will introduce obvious numerical errors.

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

$$J^{h-}_{xi,j,k} \approx -\frac{2}{\Delta x}[\lambda_{i-1,j,k}(E_*)E_* - \lambda_{i-1,j,k}(E^h_{i-1,j,k})E^h_{i-1,j,k}], \quad (12a)$$

$$J^{h+}_{xi,j,k} \approx -\frac{2}{\Delta x}[\lambda_{i,j,k}(E^h_{i,j,k})E^h_{i,j,k} - \lambda_{i,j,k}(E_*)E_*]. \quad (12b)$$

Since energy is conserved across the material interface, it must be true that $J^{h-}_{xi,j,k} = J^{h+}_{xi,j,k}$. If we temporarily assume that λ is independent of E, through this equation we may solver for E_*. After putting E_* back into Eq.(12a) or Eq.(12b), and we may get

$$J^h_{xi} \approx -\frac{\lambda_{xi,j,k}}{\Delta x}[E^h_{i,j,k} - E^h_{i-1,j,k}]. \quad (13)$$

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

$$\lambda_{xi,j,k} \equiv \frac{2\lambda_{i-1,j,k}\lambda_{i,j,k}}{\lambda_{i-1,j,k} + \lambda_{i,j,k}}. \quad (14)$$

Equation (13) with Eq.(14) is valid for different materials across the interface, which is the extension of Eq.(11).

The third issue is the nonlinearity of Eq.(1), i.e, $\lambda(E)$ is a function of E. For example, if we assume

$$\lambda(E) = \lambda^{(0)} + \lambda^{(1)}E \quad (15),$$

From the equation $J_{xi,j,k}^{h-} = J_{xi,j,k}^{h+}$ and Eqs.(12a,b), we have

$$[\lambda_{i-1,j,k}^{(0)} + \lambda_{i-1,j,k}^{(1)} E_*] E_* - \lambda_{i-1,j,k}(E_{i-1,j,k}^h) E_{i-1,j,k}^h = \lambda_{i,j,k}(E_{i,j,k}^h) E_{i,j,k}^h - [\lambda_{i,j,k}^{(0)} + \lambda_{i,j,k}^{(1)} E_*] E_*.$$

Solving for E_*, we get

$$E_* = \frac{\lambda_{i-1,j,k}^{(0)} + \lambda_{i,j,k}^{(0)}}{2[\lambda_{i-1,j,k}^{(1)} + \lambda_{i,j,k}^{(1)}]} \{[1 + \frac{4(\lambda_{i-1,j,k}^{(1)} + \lambda_{i,j,k}^{(1)})}{(\lambda_{i-1,j,k}^{(0)} + \lambda_{i,j,k}^{(0)})^2} (\lambda_{i-1,j,k} E_{i-1,j,k}^h + \lambda_{i,j,k} E_{i,j,k}^h)]^{\frac{1}{2}} - 1\}. \quad (16)$$

This formula will reduce to the following as a special case with $\lambda^{(1)}$ vanishing,

$$E_* = \frac{1}{\lambda_{i-1,j,k}^{(0)} + \lambda_{i,j,k}^{(0)}} [\lambda_{i-1,j,k} E_{i-1,j,k}^h + \lambda_{i,j,k} E_{i,j,k}^h]. \quad (17)$$

Substituting Eq.(16) into Eq.(12a,b), we may write energy flux in terms of $E_{i-1,j,k}^h$ and $E_{i,j,k}^h$. Similarly, we may find fluxes, J_{yj}^h and J_{zk}^h at the material interfaces.
Applying the fluxes in Eq.(13) to Eq.(10), and applying the result to Eqs.(5,7), we arrive at the following set of equations for $E_{i,j,k}^n$ and $E_{i,j,k}^h$,

$$E_{i,j,k}^n = E_{i,j,k} + s^h \Delta t + D_{i,j,k}^h, \quad (18a)$$

$$E_{i,j,k}^h = E_{i,j,k} + (\frac{3}{4} s^h - \frac{1}{4} s^n) \Delta t + \frac{3}{4} D_{i,j,k}^h - \frac{1}{4} D_{i,j,k}^n. \quad (18b)$$

Here $D_{i,j,k}^h$ is defined as

$$\begin{aligned} D_{i,j,k}^h \equiv & \frac{\Delta t}{(\Delta x)^2} [\lambda_{xi,j,k} E_{i-1,j,k}^h + \lambda_{xi+1,j,k} E_{i+1,j,k}^h - (\lambda_{xi,j,k} + \lambda_{xi+1,j,k}) E_{i,j,k}^h] \\ & + \frac{\Delta t}{(\Delta y)^2} [\lambda_{yi,j,k} E_{i,j-1,k}^h + \lambda_{yi,j+1,k} E_{i,j+1,k}^h - (\lambda_{yi,j,k} + \lambda_{yi,j+1,k}) E_{i,j,k}^h] \\ & + \frac{\Delta t}{(\Delta z)^2} [\lambda_{zi,j,k} E_{i,j,k-1}^h + \lambda_{zi,j,k+1} E_{i,j,k+1}^h - (\lambda_{zi,j,k} + \lambda_{zi,j,k+1}) E_{i,j,k}^h]. \quad (19) \end{aligned}$$

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

The numerical scheme, Eqs.(18a,b), 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.

We would like to point out three special cases of Eqs.(18a,b). 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

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

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

The second special case is for steady states, for which Eqs.(18a,b) become

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

This equation will give steady states for systems with different materials. The third special case is for steady states with a single material, for which the scheme is simplified to

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

Now let us discuss how to solve the large linear system, Eqs.(18a,b). The system may be iteratively solved through many approaches in [1], but our approach is simple and straightforward. It seems that we may use the evaluation of the right hand sides of Eqs.(18a,b) through an initial guess of E at $t = \Delta t$ and $t = \Delta t / 2$ to find an improved the temperatures $E^n_{i,j,k}$ and $E^h_{i,j,k}$. Unfortunately, this iterative approach doesn't converge when $\lambda \Delta t / (\Delta x)^2$ is larger than 1.

To find a successful iterative approach, we move the terms with $E^h_{i,j,k}$ and $E^h_{i,j,k}$ to the left side and write Eqs.(18a,b) in the form

$$E^n_{i,j,k} + \xi_{i,j,k} E^h_{i,j,k} = E_{i,j,k} + s^h_{i,j,k} \Delta t + Q^h_{i,j,k}, \qquad (20a)$$

$$-\frac{1}{4}\xi_{i,j,k} E^n_{i,j,k} + (1 + \frac{3}{4}\xi_{i,j,k}) E^h_{i,j,k} = E_{i,j,k} + (\frac{3}{4}s^h_{i,j,k} - \frac{1}{4}s^n_{i,j,k})\Delta t + \frac{3}{4}Q^h_{i,j,k} - \frac{1}{4}Q^n_{i,j,k}. \qquad (20b)$$

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

$$\xi_{i,j,k} \equiv \frac{\Delta t}{(\Delta x)^2}(\lambda_{xi,j,k} + \lambda_{xi+1,j,k}) + \frac{\Delta t}{(\Delta y)^2}(\lambda_{yi,j,k} + \lambda_{yi,j+1,k}) + \frac{\Delta t}{(\Delta z)^2}(\lambda_{zi,j,k} + \lambda_{zi,j,k+1}), \quad (21)$$

$$Q^h_{i,j,k} \equiv \frac{\Delta t}{(\Delta x)^2}(\lambda_{xi,j,k}E^h_{i-1,j,k} + \lambda_{xi+1,j,k}E^h_{i+1,j,k}) + \frac{\Delta t}{(\Delta y)^2}(\lambda_{yi,j,k}E^h_{i,j-1,k} + \lambda_{yi,j+1,k}E^h_{i,j+1,k}) + \frac{\Delta t}{(\Delta z)^2}(\lambda_{zi,j,k}E^h_{i,j,k-1} + \lambda_{zi,j,k+1}E^h_{i,j,k+1}), \quad (22a)$$

$$Q^n_{i,j,k} \equiv \frac{\Delta t}{(\Delta x)^2}(\lambda_{xi,j,k}E^n_{i-1,j,k} + \lambda_{xi+1,j,k}E^n_{i+1,j,k}) + \frac{\Delta t}{(\Delta y)^2}(\lambda_{yi,j,k}E^n_{i,j-1,k} + \lambda_{yi,j+1,k}E^n_{i,j+1,k}) + \frac{\Delta t}{(\Delta z)^2}(\lambda_{zi,j,k}E^n_{i,j,k-1} + \lambda_{zi,j,k+1}E^n_{i,j,k+1}). \quad (22b)$$

We solve Eqs.(20a,b) for $E^n_{i,j,k}$ and $E^h_{i,j,k}$,

$$E^n_{i,j,k} = \frac{1}{A}[(1 - \frac{1}{4}\xi_{i,j,k})E_{i,j,k} + (s^h_{i,j,k} + \frac{1}{4}\xi_{i,j,k}s^n_{i,j,k})\Delta t + Q^h_{i,j,k} + \frac{1}{4}\xi_{i,j,k}Q^n_{i,j,k}], \quad (23a)$$

$$E^h_{i,j,k} = \frac{1}{A}[(1 + \frac{1}{4}\xi_{i,j,k})E_{i,j,k} + \frac{1}{4}(3 + \xi_{i,j,k})s^h_{i,j,k}\Delta t - \frac{1}{4}s^n_{i,j,k}\Delta t + \frac{1}{4}(3 + \xi_{i,j,k})Q^h_{i,j,k} - \frac{1}{4}Q^n_{i,j,k}]. \quad (23b)$$

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.(23a,b). 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 which numerical cells are divided into two staggered sets, red and black. The two sets are alternatively updated through iterations.

4. Nonlinearity in Iterative Solvers

For the nonlinearity, classical solution techniques for radiation diffusion use a linearized radiative conductivity to avoid the expense of solving a system of nonlinear equations at each time step. This introduces a first order error in time, precluding effective use of higher order time integration methods, and small time steps must be used to control the size of this error, as shown in [12]. To avoid the error, the Newton iteration could be used for the nonlinearity, but the Newton iteration in implicit methods is very expensive.

We intent to combine the iterations for nonlinearity and implicitness together through using

the most recent values of unknowns for the evaluation of radiative conductivity during the iteration for the implicitness, but we didn't have enough time to implement this approach during the current phase of the project. Instead, in the simulations involving the nonlinearity, we have used the values of radiation energy at the previous time step to evaluate the radiative conductivity. This approach also limited the accuracy in time when the nonlinearity is involved. We plan to address this issue in future investigations.

5. Multi-grid Methods for Fast Convergence

In this section we apply the multigrid method [3,4,7] to approximately solve Eqs.(23a,b). From numerical experiments to iteratively solve Eq.(1) for a constant radiative conductivity, it is known 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, i.e., we may use the multigrid method to speed up the convergence.

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 on 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.(23a,b),

$$c^n_{i,j,k} \equiv E^n_{i,j,k} - e^n_{i,j,k}, \qquad (24a)$$

$$c^h_{i,j,k} \equiv E^h_{i,j,k} - e^h_{i,j,k}, \qquad (24b)$$

$$r^n_{i,j,k} \equiv E^n_{i,j,k} + \xi_{i,j,k} E^h_{i,j,k} - [E_{i,j,k} + s^h_{i,j,k}\Delta t + Q^h_{i,j,k}], \qquad (25a)$$

$$r^h_{i,j,k} \equiv -\frac{1}{4}\xi_{i,j,k} E^n_{i,j,k} + (1 + \frac{3}{4}\xi_{i,j,k}) E^h_{i,j,k} - [E_{i,j,k} + (\frac{3}{4} s^h_{i,j,k} - \frac{1}{4} s^n_{i,j,k})\Delta t + \frac{3}{4} Q^h_{i,j,k} - \frac{1}{4} Q^n_{i,j,k}]. \qquad (25b)$$

Here $e^n_{i,j,k}$ and $e^h_{i,j,k}$ are the exact solution of Eqs.(23a,b). Substituting $e^n_{i,j,k} = E^n_{i,j,k} - c^n_{i,j,k}$ and $e^h_{i,j,k} = E^h_{i,j,k} - c^h_{i,j,k}$ into Eqs.(23a,b), we have

$$c^n_{i,j,k} + \xi_{i,j,k} c^h_{i,j,k} = Q^h_{i,j,k}(c), \qquad (26a)$$

$$-\frac{1}{4}\xi_{i,j,k}c^{n}_{i,j,k}+(1+\frac{3}{4}\xi_{i,j,k})c^{h}_{i,j,k}=\frac{3}{4}Q^{h}_{i,j,k}(c)-\frac{1}{4}Q^{n}_{i,j,k}(c). \qquad (26b)$$

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.(22a,b) but are evaluated at $c^{n}_{i,j,k}$ and $c^{h}_{i,j,k}$ instead of $E^{n}_{i,j,k}$ and $E^{h}_{i,j,k}$. Eqs.(26a,b) may be iteratively solved in the exactly same way as we do with Eqs.(23a,b). A few iterations of Eqs.(26a,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 a 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 $E^{n}_{i,j,k}$ and $E^{h}_{i,j,k}$,

$$E^{n}_{i,j,k}=E^{n}_{i,j,k}-c^{n}_{i,j,k},$$
$$E^{h}_{i,j,k}=E^{h}_{i,j,k}-c^{h}_{i,j,k}.$$

This procedure is 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 $E^{n}_{i,j,k}$ and $E^{h}_{i,j,k}$ 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 $E^{n}_{i,j,k}$ and $E^{h}_{i,j,k}$ defined on a set of fine cells together in order to find cell-averaged $E^{n}_{i,j,k}$ and $E^{h}_{i,j,k}$ on the coarser grid. For the mapping from the coarser grid to the 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 structures within 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. As a result, there will be no difference in the convergence rate between the different interpolations for the mapping. Therefore, we use the piecewise constant interpolation for the mapping.

6. Numerical Examples

In this section we will provide numerical examples to show the correctness and the features of the schemes for single and multi-materials. The first example is to show how the solutions of the scheme change with different time steps. Figure 3 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. As expected, the numerical solution after a very large time step is the correct steady state.

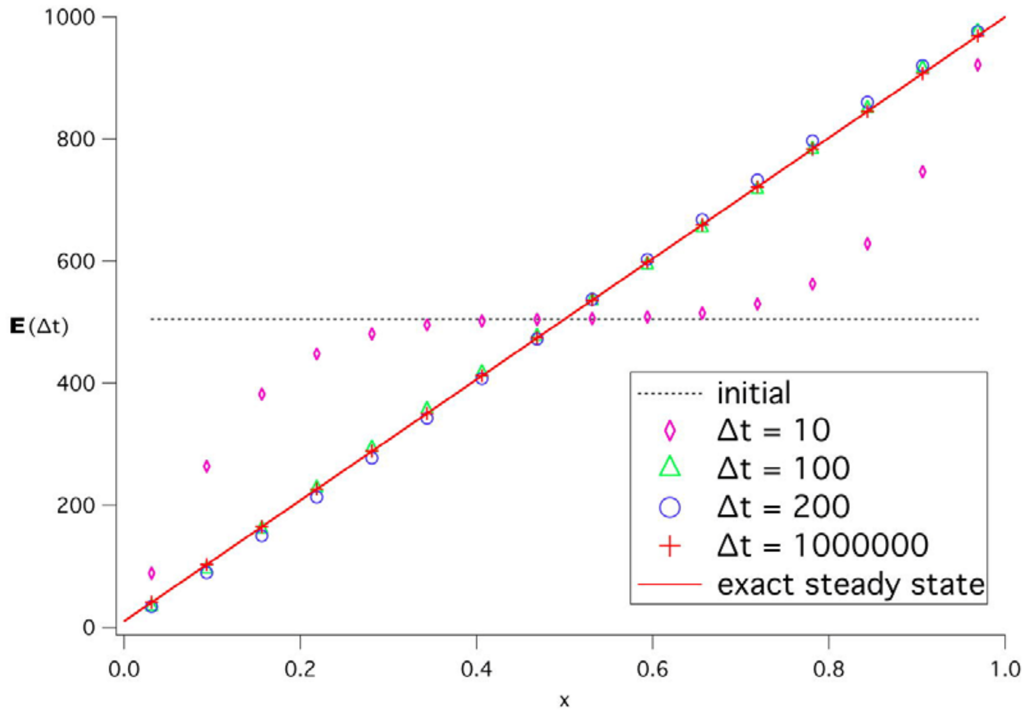

Figure 3. Four numerical solutions after one time step with four sizes. The dashed line is the initial condition, and the solid line is the exact steady state. Sixteen grid cells are used in these one-dimensional simulations.

The second problem is to show the correctness when two materials are involved. Figure 4 shows a solution after a very large time step of a one-dimensional simulation with 48 grid cells, fixed energy at boundaries, and two materials, one with the radiative conductivity 0.01 ($x < 0.25$ and $x > 0.75$) and the other with radiative conductivity 0.001 ($0.25 < x > 0.75$). The dashed line in the figure is the initial energy and the solid line is the exact steady state of the problem. The red dots are the solutions of our approach using Eq.(14) for the interfaces between the two materials. The blue dots are the results obtained through Eq.(11b). It should be pointed out that dT/dx is discontinuous at x = 0.25 and 0.75, but the 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 accurate for problems involving multi-materials, while the typical previous methods, for example Eq.(11b), resulting obvious numerical errors near the discontinuities.

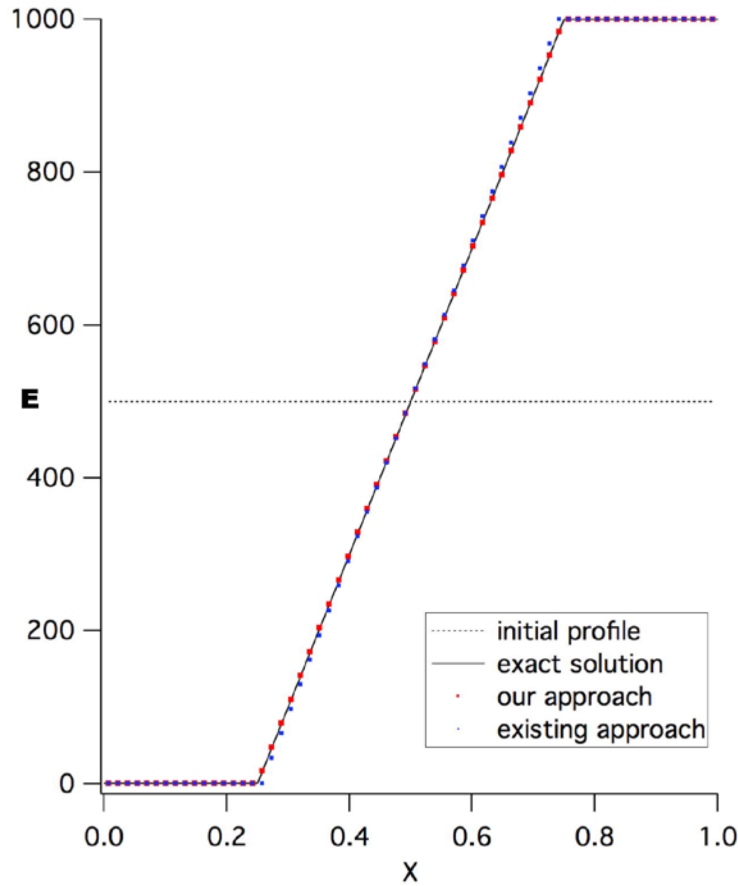

Figure 4. The solution after a very large time step for one-dimensional problem with two materials. Fourty eight grid cells are used. The dashed line is the initial condition, and the solid time is the exact steady state. The red dots are the solutions of our method, while the blue dots are obtained from an existing typical approach that conatins obvious numerical errors near the discontinuities.

The next two examples are to show the effectiveness of multigrid method in two- and three-dimensional cases. Figure 5 shows the convergence rates of the red-black and multigrid methods 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. 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.6. The actual values at the end of multigrid method are (0.09, 2.9e-07).

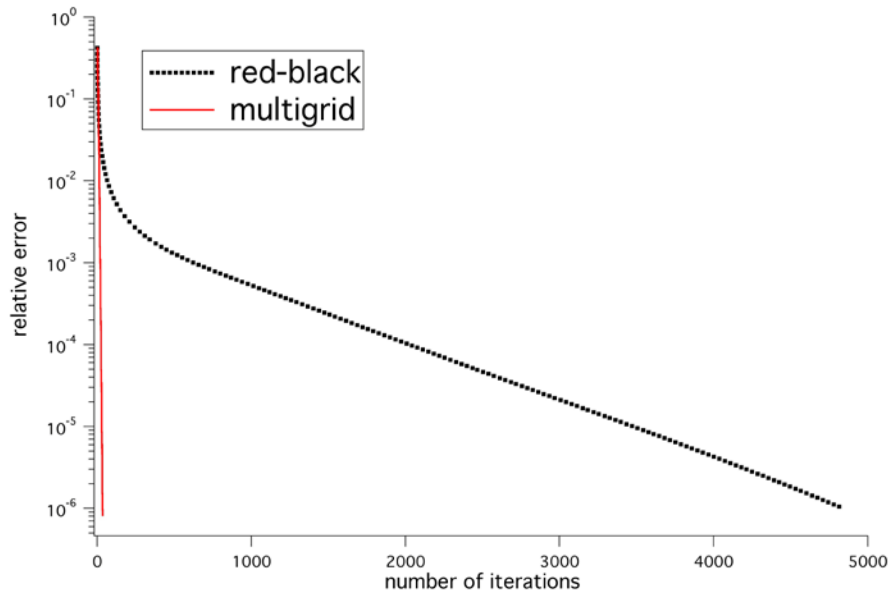

Figure 5. The converence of two iterative approaches, red-black and multigrid method, in a two-dimensional simulations. $\lambda \Delta t/(\Delta x)^2 = 1638.4$.The actual number of iteration for the multgrid method is 24 at the end of the red line.

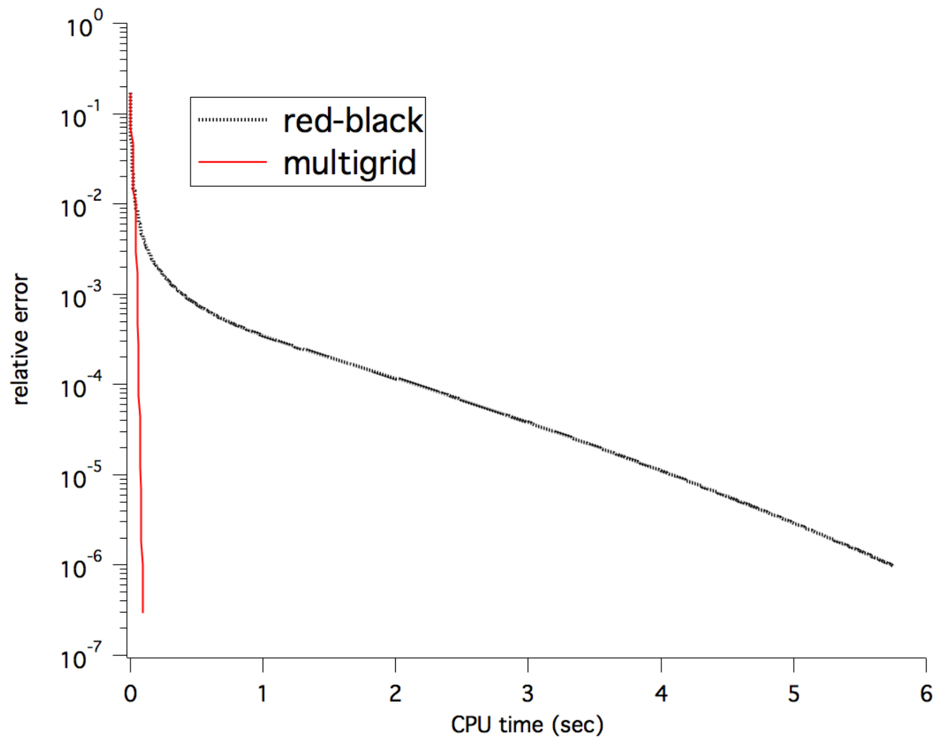

Figure 6. Relative errors vs the CPU time in two iterative approaches, red-black and multigrid, in a two-dimensional simulation. $\lambda \Delta t/(\Delta x)^2 = 1638.4$. The actual CPU time for the multigrid method at the end of the red line is 0.09 sec.

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. 7 and 8 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.7 and 8 are 36 and 35.25 sec.

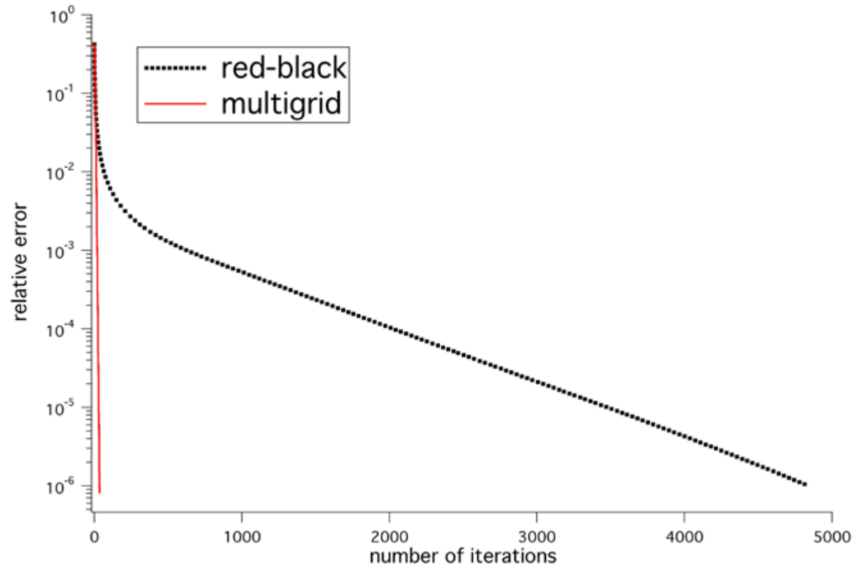

Figure 7. The converence of two iterative approaches, red-black and multigrid method, in a three-dimensional simulations. $\lambda \Delta t/(\Delta x)^2$ = 1638.4. The actual number of iteration for the multgrid method is 36 at the end of the red line.

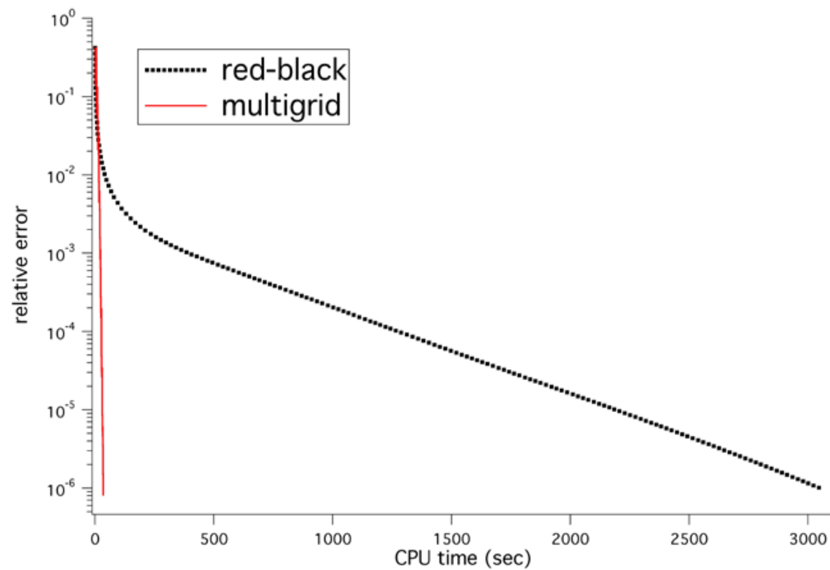

Figure 8. Relative errors vs the CPU time in two iterative approaches, red-black and multigrid, in a three-dimensional simulation. $\lambda \Delta t/(\Delta x)^2$ = 1638.4. The actual CPU time for the multigrid method at the end of the red line is 35 sec.

7. Simulation for a Nuclear Accident

In this section, we will present a two-dimensional simulation in a square domain for a nuclear accident that involves leaking of radiation energy. The simulation is performed in a square domain 10×10 with 1000×1000 grid cells. A square region at the center of simulation domain is full of radiation energy at the level 10^5 while radiation energy is unit in the other part of simulation domain. Surrounding the radiation energy, there are two layers of material which prevent radiation from diffusing away from the center region. We assume that the radiative conductivity of all materials has the form

$$\lambda(E) = \lambda_0(1 + \varepsilon E).$$

Here ε is 10^{-4}. The radiative conductivity λ_0 is assumed to be 1 for air, and 10^{-3} and 10^{-4} for the two protection layers. The size of time step is set to 0.004 and the maximum value of $\lambda \Delta t / (\Delta x)^2$ is around 40. As stated before, the value of radiation energy E at the previous time step is used to calculate the value of radiative conductivity, since iterative solver for the set of fully nonlinear algebraic eauations remains to be develeloped in this project.

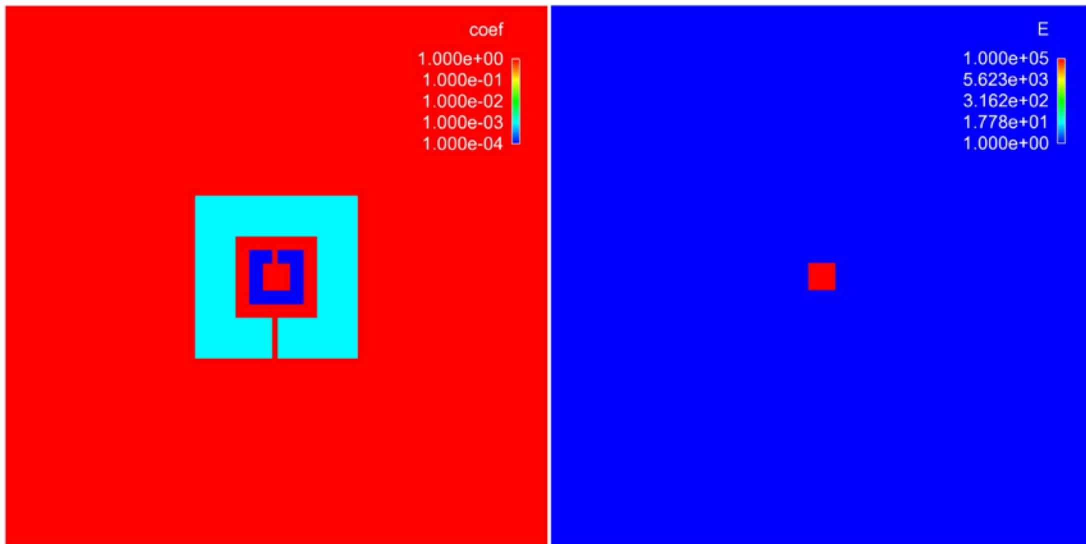

Figure 9. A two-dimensional simulation with 500 x 500 grid cells on 10 x 10 domain. The left image is the values of radiative conductivity, in which there are two layers of material with very low radiative conductivity. The image at the right is the initial distribution of radiation energy.

Figure 9 shows the initial values of radiative conductivity and radiation energy. The left image is the values of radiative conductivity, in which there are two layers of material with very

low radiative conductivity. In the image there are two cracks in the two protection materials, which represent the cracks in a nuclear accident and cuase radiation energy to leak away from the center region. In Fig.10, we display six instants when radiation energy diffuses away from the center region.

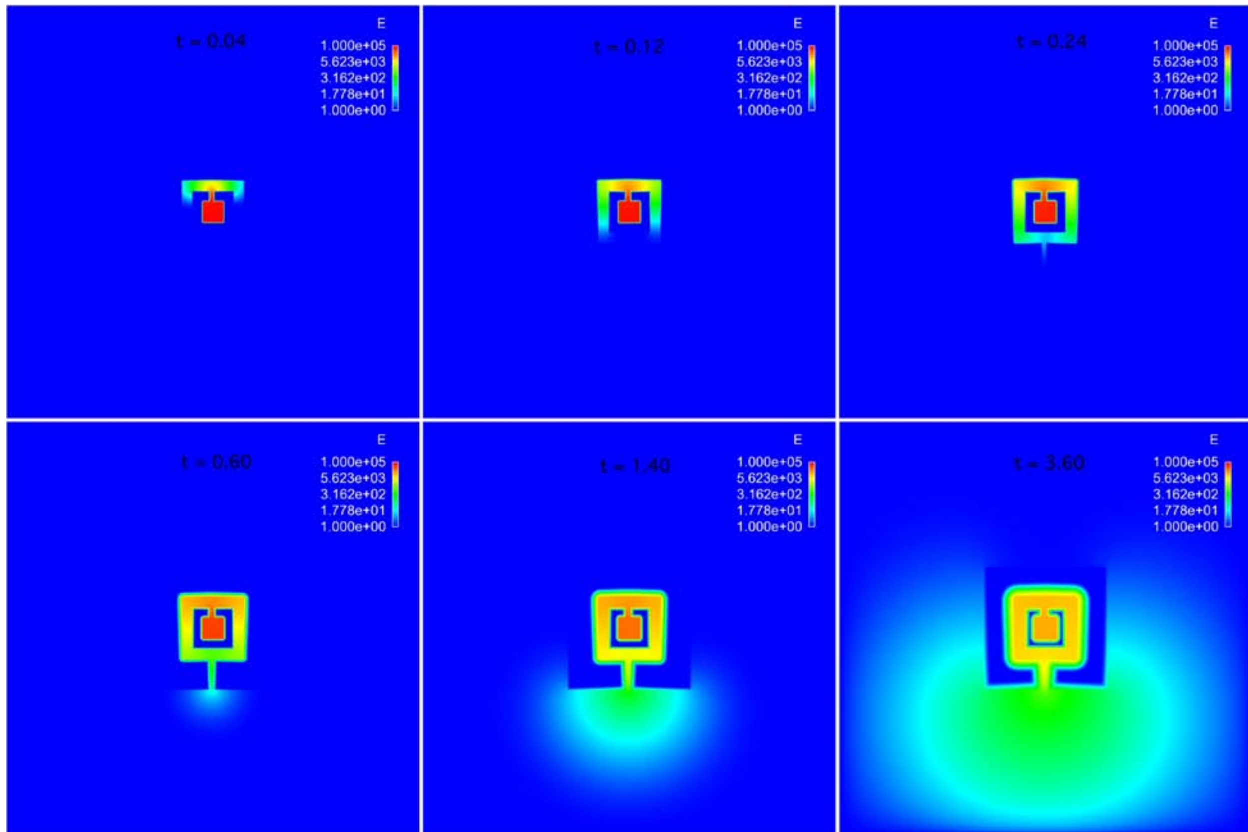

Figure 10. Six instants of a two-dimensional simulation for leaking of radiation energy.

8. Conclusions

We have developed computer model, a numerical scheme, and computer codes for radiation diffusion in nuclear accidents. The development of the numerical method involve several difficulties, which include accuracy of numerical methods, correct behavior when $\lambda\Delta t/(\Delta x)^2$ is large, the discontinuity of radiative conductivity between materials, the nonlinearity of the basic equation.

The discontinuity of radiative conductivity between different materials has been correctly treated. The effective radiative conductivity for an interface of any two materials derived in this project is based on physics principles. The method is second order accurate in both space and

time. When the time step is very large, the scheme gives correct steady states. An efficient iterative approach to solve the set of algebraic equations arising from the implicitness of the scheme has been developed through the multigrid method. The features of the scheme have been demonstrated through numerical examples in one-, two-, and three-dimensional situations, which involve dramatically different materials. Although we have not finished any numerical simulations for real nuclear accident, a simple simulation involves a generic situation has been demonstrated.

We intent to develop an advanced technique to iteratively solve the nonlinearity and implicitness together, but we didn't have enough time to implement this approach during the current phase of the project. Instead, in the simulations involving the nonlinearity, we have used the values of radiation energy at the previous time step to evaluate the radiative conductivity. An iterative solver for a system of fully nonlinear equations remains to be investigated.

9. Acknowledgment

We deeply appreciate our teacher, Lee Goodwin, for his support and help, our mentor, William Dai, for his direction, suggestion, and help during the project, and Professor Paul Woodward for his direction and suggestion for computational physics.

References

[1] D.M. Young, Iterative Solution of Large Linear Systems (Academic Press, New York, 1971).

[2] 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).

[3] P. Wesseling, Theroretical and practical aspects of a multigrid method, SIAM J. Sci. Comput. 3, 387 (1982).

[4] W. Hackbusch, Multi-grid methods and applications (Springer, Berlin, 1985).

[5] 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).

[6] Y. Jaluria and K. E. Torrance, Computational Heat Transfer (Spring-Verlag, Berlin, 1986).

[7] S. F. McComick, Multigrid methods (Frontier in Applied Mathematics), Vol.3 (SIAM, Philadelphia, 1989).

[8] Staff, IAEA, AEN/NEA (in Technical English). [www-pub.iaea.org/MTCD/publications/PDF/INES-2009_web.pdf International Nuclear and Radiological Events Scale Users' Manual, 2008 Edition]. Vienna, Austria: International Atomic Energy Agency. p. 184. www-pub.iaea.org/MTCD/publications/PDF/INES-2009_web.pdf. Retrieved 2010-07-26.

[9] Chernobyl Final Report, “The Human Consequences of the Chernobyl Nuclear Accident”. UNDP and UNICEF. 22 January 2002.

[10] BBC LIVE: Japan.

[11] Benjamin K. Sovacool. A Critical Evaluation of Nuclear Power and Renewable Electricity in Asia, Journal of Contemporary Asia, Vol. 40, No. 3, August 2010, pp. 393–400.

[12] C. C. Ober and J. N. Shadid, Studies on the accuracy of time integration methods for the radiation diffusion equations, J. Comput. Phys., 195 (2004), pp. 743–772.

[13] D. Mihalas and B. Weibel-Mihalas, Foundations of Radiation Hydrodynamics, Dover Publications, Inc., Mineola, NY, 1999.

[14] E. Minguez, P. Martel, M. Gil, J. G. Rubiano, and R. Rodriguez, Analytic opacity formulas for ICF elements, Fusion Eng. Des., 60 (2002), pp. 17–25.

APPENDIX Source Codes

```
#ifndef RADIATION_H
#define RADIATION_H

#ifdef __cplusplus
extern "C" {
#endif

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        **************************************************
        *****              radiation.h               *****
        *****       THIS FILE WAS WRITTEN BY         *****
        *****                Team 66                 *****
        *****              April 2010                *****
        **************************************************
 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
#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];
    double coordr[3];

    /* old radiation energy, exist only for finest mesh */
    double *e1d;        /* 1D radiation energy, valid only for 1D case */
    double **e2d;       /* 2D radiation energy, valid only for 2D case */
    double ***e3d;      /* 3D radiation energy, valid only for 3D case */

    /* radiation energy after one time step */
    double *en1d;       /* 1D radiation energy, valid only for 1D case */
    double **en2d;      /* 2D radiation energy, valid only for 2D case */
    double ***en3d;     /* 3D radiation energy, valid only for 3D case */

    /* radiation energy after a half time step */
    double *eh1d;       /* 1D radiation energy, valid only for 1D case */
    double **eh2d;      /* 2D radiation energy, valid only for 2D case */
    double ***eh3d;     /* 3D radiation energy, 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 */

    /* radiative 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;
#ifdef __cplusplus
}
#endif
#endif
```

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        ***********************************************
        *****              radiation.c              *****
        *****        THIS FILE WAS WRITTEN BY       *****
        *****               Team 66                 *****
        *****              April 2010               *****
        ***********************************************
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

#include "radiation.h"
#include "mio.h"

const int dim = 2;
const int if_multigrid = 1;

double simulation_time = 0.0;
double dt_src       = 0.1;  /* src0 takes effect only when simulation_time < dt_src */
double src0         = 0.0;

double coef_epsilon    = 0.00001;          /* c = c0 * (1 + coef_epsilon * E)     */

double ebdry[] = {1, 1.0, 1.0, 1.0, 1.0, 1.0};  /* bdry_el, bdry_er, bdry_en, bdry_ef, bdry_eb,
bdry_et */
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 = 0.0;      /* initial energy  */

double gcoordl[] = {0.0, 0.0, 0.0};  /* the lower ends of simulation domain */
double gcoordr[] = {10.0, 10.0, 10.0};  /* the high ends of simulation domain  */
int    gsizes[]  = {500, 500, 500};  /* 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[] = {0.000001, 0.1};
double mat_coordl_array[3][3] = {{-1.0, -1.0, -1.0}, {0.125, 0.125, 0.125}};
double mat_coordr_array[3][3] = {{ 2.0,  2.0,  2.0}, {0.875, 0.875, 0.875}};
***/

// five materials
int nmat = 7;
double coef_array[]   = {1.0, 0.001, 1.0, 0.0001, 1.0,     1.0, 1.0};
double energy_array[] = {1.0, 1.0,   1.0, 1.0,    100000.0, 1.0, 1.0};

double mat_coordl_array[7][3] = {{-10.0, -10.0, -10.0},           /* cover the simulation domain */
                                 {  3.5,   3.5,   3.5},
                                 {  4.25,  4.25,  4.25},
                                 {  4.5,   4.5,   4.5 },
                                 {  4.75,  4.75,  4.75},
                                 {  4.95,  5.25,  5.125},
                                 {  4.95,  3.5,   4.0}
                                };
double mat_coordr_array[7][3] = {{20.0,   20.0,   20.0},
                                 { 6.5,    6.5,    6.5},
                                 { 5.75,   5.75,   5.75},
                                 { 5.5,    5.5,    5.5},
                                 { 5.25,   5.25,   5.25},
```

```
                        { 5.05,   5.5,    5.5},
                        { 5.05,   4.25,   4.25}
                        };

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;
int niterf = 20;
int niterc = 20;
double afine = 0.0;
double acoar = 0.0;
double small = 1.0e-10;


int    niter_max = 256;          /* max number of iteration for move_down */
double accuracy  = 1.0e-06;
double dt    = 0.004;            /* time step         */
double tstop = 200.0;            /* time to terminate  */
int    nstop = 1000000000;

double dtdump = 100000.0;
int    ndump  = 1;

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 energy_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_radiative_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, *child;

t = 0.0;
simulation_time = t;
ncycle = 0;
idump  = 0;
tdump  = 0;
todump = ndump;

energy_init(&mesh, gcoordl, gcoordr, gsizes);
set_gause_1d(&mesh);

dump(&mesh, t, idump);

idump++;

while ((t < tstop) && (ncycle < nstop) ) {
   set_radiative_coef(&mesh);
   child = mesh.child;
   if (child) {
      set_coef_fr_parent(child);
   }
   courant = dt * coefbydc2;
   printf("ncycle = %d, t = %e, courant = %e\n", ncycle, t, courant);
   initial_guess(&mesh);
   t0_in_timer = clock();;
   multigrid(&mesh, dt);
   update(&mesh);

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

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

int multigrid(H_Mesh *m, double dt)
{
    int niter_already = 0;
    double err;

    niter_already = 0;
    err = 1.0;
    while (err > accuracy) {
        move_dn(m, dt, niter_max, &err, &niter_already);
        if (if_multigrid) {
            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)
{
    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);
            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);
            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);
                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)
{
    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) {
```

```
            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) {
            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) {
            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 energy_init(H_Mesh *mesh, double *gcoordl, double *gcoordr, int *sizes)
{
    int d, to_coarsen;
    int hsizes[3];
    H_Mesh *parent, *child;

    set_mesh(gcoordl, gcoordr, sizes, mesh);
    set_radiative_coef(mesh);
    set_initial_temp(mesh);
    set_bdry(mesh, ebdry);
    if (if_multigrid == 0) return 0;

    parent = mesh;
    check_ifcoarse(parent, &to_coarsen);
    while (to_coarsen) {
        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);
        child->parent = parent;
        child->level = parent->level + 1;
        parent->child = child;
        set_coef_fr_parent(child);
        parent = child;
        check_ifcoarse(parent, &to_coarsen);
        if (hsizes[0] <= 4) {
            to_coarsen = 0;
```

```
        }
    }
    return 0;
}

int map_up1d(H_Mesh *parent, double dt)
{
    int *sizes, i0, i, ic;
    double *th, *tn, *thc, *tnc;
    H_Mesh *child;

    assert(parent);
    sizes = parent->sizes;
    th = parent->eh1d;
    tn = parent->en1d;
    child = parent->child;
    assert(child);
    thc = child->eh1d;
    tnc = child->en1d;
    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)
{
    int *sizes, i0, i, ic, j0, j, jc;
    double **th, **tn, **thc, **tnc;
    H_Mesh *child;

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

```
    th = parent->eh2d;
    tn = parent->en2d;
    child = parent->child;
    assert(child);
    thc = child->eh2d;
    tnc = child->en2d;
    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)
{
    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->eh3d;
    tn = parent->en3d;
    child = parent->child;
    assert(child);
    thc = child->eh3d;
    tnc = child->en3d;
    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)
{
    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)
{
    int i, j, k, *sizes;
    double *e1d, *eh1d, *en1d, **e2d, **eh2d, **en2d;
    double ***e3d, ***eh3d, ***en3d;

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

    if (dim == 1) {
```

```
        eh1d = mesh->eh1d;
        en1d = mesh->en1d;
        e1d  = mesh->e1d;
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            e1d[i] = en1d[i];
            eh1d[i] = en1d[i];
        }
    }
    else if (dim == 2) {
        eh2d = mesh->eh2d;
        en2d = mesh->en2d;
        e2d  = mesh->e2d;
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                e2d[j][i] = en2d[j][i];
                eh2d[j][i] = en2d[j][i];
            }
        }
    }
    else if (dim == 3) {
        eh3d = mesh->eh3d;
        en3d = mesh->en3d;
        e3d  = mesh->e3d;
        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++) {
                    e3d[k][j][i] = en3d[k][j][i];
                    eh3d[k][j][i] = en3d[k][j][i];
                }
            }
        }
    }
    return 0;
}

int initial_guess(H_Mesh *mesh)
```

```
{
    int i, j, k, *sizes;
    double *e1d, *eh1d, *en1d, **e2d, **eh2d, **en2d;
    double ***e3d, ***eh3d, ***en3d;
    H_Mesh *child;

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

    if (dim == 1) {
        eh1d = mesh->eh1d;
        en1d = mesh->en1d;
        e1d  = mesh->e1d;
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            en1d[i] = e1d[i];
            eh1d[i] = e1d[i];
        }
    }
    else if (dim == 2) {
        eh2d = mesh->eh2d;
        en2d = mesh->en2d;
        e2d  = mesh->e2d;
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                en2d[j][i] = e2d[j][i];
                eh2d[j][i] = e2d[j][i];
            }
        }
    }
    else if (dim == 3) {
        eh3d = mesh->eh3d;
        en3d = mesh->en3d;
        e3d  = mesh->e3d;
        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++) {
                    en3d[k][j][i] = e3d[k][j][i];
                    eh3d[k][j][i] = e3d[k][j][i];
                }
            }
        }
    }
    child = mesh->child;
    while (child) {
        initial_guess0(child);
        child = child->child;
    }
    return 0;
}

int initial_guess0(H_Mesh *mesh)
{
    int i, j, k, *sizes;
    double *e1d, *eh1d, *en1d, **e2d, **eh2d, **en2d;
    double ***e3d, ***eh3d, ***en3d;

    sizes = mesh->sizes;
    if (dim == 1) {
        eh1d = mesh->eh1d;
        en1d = mesh->en1d;
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            en1d[i] = 0.0;
            eh1d[i] = 0.0;
        }
    }
    else if (dim == 2) {

        eh2d = mesh->eh2d;
        en2d = mesh->en2d;
        for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
```

```
                en2d[j][i] = 0.0;
                eh2d[j][i] = 0.0;
            }
        }
    }
    else if (dim == 3) {
        eh3d = mesh->eh3d;
        en3d = mesh->en3d;
        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++) {
                    en3d[k][j][i] = 0.0;
                    eh3d[k][j][i] = 0.0;
                }
            }
        }
    }
    return 0;
 }

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

    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 */

    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 */

    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)
{
    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->e1d;
    tn  = m->en1d;
    th  = m->eh1d;
    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);
        }
        if (courant < 1000000.0) {
            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) ) );
    }
  }
  else {
    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];
        etadtbyc = etadt * courinv;
        aa   = courinv2 + 0.25 * etadtbyc * (3.0 * courinv + etadtbyc);
        ainv = 1.0 /aa;
        qh   = tmp[0] *(cx[i] * th[i-1] + cx[i+1] * th[i+1]);
        qn   = tmp[0] *(cx[i] * tn[i-1] + cx[i+1] * tn[i+1]);
        srcn = (courinv + 0.25 * etadtbyc) * srcdt
             + (1.0 + 0.75 * etadt) * sn[i] /* sn[i] and sh[i] were already */
             - etadt * sh[i];               /* multiplied by courinv when obtained */
        srcn *= courinv;
        srch = (0.5 * courinv + 0.25 * etadtbyc) * srcdt
             + (0.25 * etadt * sn[i] + sh[i]);
        srch *= courinv;
        tni  = ((courinv - 0.25 * etadtbyc) * courinv * t[i]
             + srcn + courinv * qh + 0.25 * etadtbyc * qn) * ainv;
        thi  = ((courinv + 0.25 * etadtbyc) * courinv * t[i]
             + srch + (0.75 * courinv + 0.25 * etadtbyc) * qh
```

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

            *err_out = MAX(*err_out, MAX(fabs(rn[i]), fabs(rh[i]) ) );

           /* sn[i] and sh[i] are already multiplied by courinv when obtained */
         }
      }
      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 solver2d(H_Mesh *m, double dt, int niter_allowed, double accuracy_req, double *err_out, int
*niter_already)
{
    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->e2d;
    tn  = m->en2d;
    th  = m->eh2d;
    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);
   }
   if (courant < 10000000.0) {    /* for the time steps that are not infinite */
      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) ) );
        }
      }
    }
    else {
      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] + cx[j+1][i]) * dtbydc2[1];
            etadtbyc = etadt * courinv;
            aa   = courinv2 + 0.25 * etadtbyc * (3.0 * courinv + etadtbyc);
            ainv = 1.0 /aa;
            qh  = tmp[0] *(cx[j][i] * th[j][i-1] + cx[j][i+1] * th[j][i+1])
                + tmp[1] *(cy[j][i] * th[j-1][i] + cy[j+1][i] * th[j+1][i]);
            qn  = tmp[0] *(cx[j][i] * tn[j][i-1] + cx[j][i+1] * tn[j][i+1])
                + tmp[1] *(cy[j][i] * tn[j-1][i] + cy[j+1][i] * tn[j+1][i]);
            srcn = (courinv + 0.25 * etadtbyc) * srcdt
                 + (1.0 + 0.75 * etadt) * sn[j][i] /* sn[i] and sh[i] were already */
                 - etadt * sh[j][i];              /* multiplied by courinv when obtained */
            srcn *= courinv;
            srch = (0.5 * courinv + 0.25 * etadtbyc) * srcdt
```

```
                + (0.25 * etadt * sn[j][i] + sh[j][i]);
            srch *= courinv;
            tni  = ((courinv - 0.25 * etadtbyc) * courinv * t[j][i]
                + srcn + courinv * qh + 0.25 * etadtbyc * qn) * ainv;
            thi  = ((courinv + 0.25 * etadtbyc) * courinv * t[j][i]
                + srch + (0.75 * courinv + 0.25 * etadtbyc) * qh
                - 0.25 * courinv * qn) * ainv;
            tn[j][i] = tni;
            th[j][i] = thi;
         }
         istart = 1 - istart;
      }
   }
   for (j0 = 0; j0 < sizes[1]; j0++) {
      j = nbdy + j0;
      for (i0 = 0; i0 < sizes[0]; i0++) {
         i = nbdy + i0;
         etadtbyc = tmp[0] * (cx[j][i] + cx[j][i+1])
                + tmp[1] * (cy[j][i] + cy[j+1][i]);
         qh  = tmp[0] *(cx[j][i] * th[j][i-1] + cx[j][i+1] * th[j][i+1])
            + tmp[1] *(cy[j][i] * th[j-1][i] + cy[j+1][i] * th[j+1][i]);
         qn  = tmp[0] *(cx[j][i] * tn[j][i-1] + cx[j][i+1] * tn[j][i+1])
            + tmp[1] *(cy[j][i] * tn[j-1][i] + cy[j+1][i] * tn[j+1][i]);

         rn[j][i] = courinv * tn[j][i] + etadtbyc * th[j][i]
                - (courinv * t[j][i] + srcdt * courinv + sn[j][i] +  qh);
         rh[j][i] = -0.25 * etadtbyc * tn[j][i] + (courinv + 0.75 * etadtbyc) * th[j][i]
                - (courinv * t[j][i] + 0.5 * srcdt * courinv + sh[j][i] + 0.75 * qh - 0.25 * qn);

         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) ) );

        /* sn[i] and sh[i] are already multiplied by courinv when obtained */
      }
   }
}
```

```
        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 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->e3d;
tn  = m->en3d;
th  = m->eh3d;
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);
    }
    if (courant < 100000000.0) {    /* for the time steps that are not infinite */
        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) )
);
          }
        }
      }
    }
    else {
      istart0 = 0;
      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];
            etadtbyc = etadt * courinv;
            aa   = courinv2 + 0.25 * etadtbyc * (3.0 * courinv + etadtbyc);
            ainv = 1.0 /aa;
            qh  = tmp[0] *(cx[k][j][i] * th[k][j][i-1] + cx[k][j][i+1] * th[k][j][i+1])
              + tmp[1] *(cy[k][j][i] * th[k][j-1][i] + cy[k][j+1][i] * th[k][j+1][i])
              + tmp[2] *(cz[k][j][i] * th[k-1][j][i] + cz[k+1][j][i] * th[k+1][j][i]);
            qn  = tmp[0] *(cx[k][j][i] * tn[k][j][i-1] + cx[k][j][i+1] * tn[k][j][i+1])
              + tmp[1] *(cy[k][j][i] * tn[k][j-1][i] + cy[k][j+1][i] * tn[k][j+1][i])
              + tmp[2] *(cz[k][j][i] * tn[k-1][j][i] + cz[k+1][j][i] * tn[k+1][j][i]);
            srcn = (courinv + 0.25 * etadtbyc) * srcdt
               + (1.0 + 0.75 * etadt) * sn[k][j][i] /* sn[i] and sh[i] were already */
               - etadt * sh[k][j][i];            /* multiplied by courinv when obtained */
            srcn *= courinv;
```

```
            srch = (0.5 * courinv + 0.25 * etadtbyc) * srcdt
                + (0.25 * etadt * sn[k][j][i] + sh[k][j][i]);
            srch *= courinv;
            tni  = ((courinv - 0.25 * etadtbyc) * courinv * t[k][j][i]
                + srcn + courinv * qh + 0.25 * etadtbyc * qn) * ainv;
            thi  = ((courinv + 0.25 * etadtbyc) * courinv * t[k][j][i]
                + srch + (0.75 * courinv + 0.25 * etadtbyc) * qh
                - 0.25 * courinv * qn) * ainv;
            tn[k][j][i] = tni;
            th[k][j][i] = thi;
        }
        istart = 1 - istart;
    }
    istart0 = 1 - istart0;
}
istart0 = 1;
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];
            etadtbyc = etadt * courinv;
            aa   = courinv2 + 0.25 * etadtbyc * (3.0 * courinv + etadtbyc);
            ainv = 1.0 /aa;
            qh  = tmp[0] *(cx[k][j][i] * th[k][j][i-1] + cx[k][j][i+1] * th[k][j][i+1])
               + tmp[1] *(cy[k][j][i] * th[k][j-1][i] + cy[k][j+1][i] * th[k][j+1][i])
               + tmp[2] *(cz[k][j][i] * th[k-1][j][i] + cz[k+1][j][i] * th[k+1][j][i]);
            qn  = tmp[0] *(cx[k][j][i] * tn[k][j][i-1] + cx[k][j][i+1] * tn[k][j][i+1])
               + tmp[1] *(cy[k][j][i] * tn[k][j-1][i] + cy[k][j+1][i] * tn[k][j+1][i])
               + tmp[2] *(cz[k][j][i] * tn[k-1][j][i] + cz[k+1][j][i] * tn[k+1][j][i]);
            srcn = (courinv + 0.25 * etadtbyc) * srcdt
```

```
          + (1.0 + 0.75 * etadt) * sn[k][j][i] /* sn[i] and sh[i] were already */
          - etadt * sh[k][j][i];              /* multiplied by courinv when obtained */
        srcn *= courinv;
        srch = (0.5 * courinv + 0.25 * etadtbyc) * srcdt
          + (0.25 * etadt * sn[k][j][i] + sh[k][j][i]);
        srch *= courinv;
        tni  = ((courinv - 0.25 * etadtbyc) * courinv * t[k][j][i]
          + srcn + courinv * qh + 0.25 * etadtbyc * qn) * ainv;
        thi  = ((courinv + 0.25 * etadtbyc) * courinv * t[k][j][i]
          + srch + (0.75 * courinv + 0.25 * etadtbyc) * qh
          - 0.25 * courinv * qn) * ainv;
        tn[k][j][i] = tni;
        th[k][j][i] = thi;
      }
      istart = 1 - istart;
    }
    istart0 = 1 - istart0;
  }
  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;
        etadtbyc = (cx[k][j][i] + cx[k][j][i+1]) * tmp[0]
            + (cy[k][j][i] + cy[k][j+1][i]) * tmp[1]
            + (cz[k][j][i] + cz[k+1][j][i]) * tmp[2];
        qh  = tmp[0] *(cx[k][j][i] * th[k][j][i-1] + cx[k][j][i+1] * th[k][j][i+1])
          + tmp[1] *(cy[k][j][i] * th[k][j-1][i] + cy[k][j+1][i] * th[k][j+1][i])
          + tmp[2] *(cz[k][j][i] * th[k-1][j][i] + cz[k+1][j][i] * th[k+1][j][i]);
        qn  = tmp[0] *(cx[k][j][i] * tn[k][j][i-1] + cx[k][j][i+1] * tn[k][j][i+1])
          + tmp[1] *(cy[k][j][i] * tn[k][j-1][i] + cy[k][j+1][i] * tn[k][j+1][i])
          + tmp[2] *(cz[k][j][i] * tn[k-1][j][i] + cz[k+1][j][i] * tn[k+1][j][i]);

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

```
                        rh[k][j][i] = -0.25 * etadtbyc * tn[k][j][i] + (courinv + 0.75 * etadtbyc) *
th[k][j][i]
                                    - (courinv * t[k][j][i] + 0.5 * srcdt * courinv + sh[k][j][i] + 0.75 * qh -
0.25 * qn);
                        /* sn[i] and sh[i] are already multiplied by courinv when obtained */
                    }
                }
            }
        }
        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)
{
   int i, i1, j, j1, k, k1;
   int *sizes;
   double *e1d, **e2d, ***e3d;

   sizes = m->sizes;
   if (dim == 1) {
       e1d = m->e1d;
       for (i = 0; i < nbdy; i++) {
           i1 = sizes[0] + nbdy + i;
           e1d[i]  = bdry[0];
           e1d[i1] = bdry[1];
```

```
        }
    }
    else if (dim == 2) {
        e2d = m->e2d;
        for (j = 0; j < nbdy; j++) {
            j1 = sizes[1] + nbdy + j;
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                e2d[j][i]  = bdry[2];
                e2d[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;
                e2d[j1][i]  = bdry[0];
                e2d[j1][i1] = bdry[1];
            }
        }
    }
    else if (dim == 3) {
        e3d = m->e3d;
        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++) {
                    e3d[k][j][i]  = bdry[4];
                    e3d[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++) {
```

```
                e3d[k1][j][i]  = bdry[2];
                e3d[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;
                e3d[k1][j1][i]  = bdry[0];
                e3d[k1][j1][i1] = bdry[1];
            }
        }
    }
  }
  return 0;
}

int set_T_bdry1d(H_Mesh *m)
{
  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->eh1d;
    tn = m->en1d;
    sloph0 = 2.0 * (th[nbdy] - ebdry[0])/dc[0];
    slopn0 = 2.0 * (tn[nbdy] - ebdry[0])/dc[0];
    sloph1 = 2.0 * (ebdry[1] - th[i1_src])/dc[0];
    slopn1 = 2.0 * (ebdry[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)
{
    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)
```

```
{
   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)
{
   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_radiative_coef(H_Mesh *m)
{
    int i, j, k, imat, d;
    int *sizes, offsets[3], msizes[3], sizes_ext[3];
    double *coordl, *coordr, *mcoordl, *mcoordr, coordl_ext[3];
    double temp, coef, coef_max, dc, sml, dcinv[3], dc2inv[3];
    double *e1d, **e2d, ***e3d, *c1d, **c2d, ***c3d;
    double *cx1d, **cx2d, **cy2d, ***cx3d, ***cy3d, ***cz3d;

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

    e1d = m->e1d;
    e2d = m->e2d;
    e3d = m->e3d;

    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];
        temp = energy_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) {
    if (simulation_time < 0.5 * dt) {
      for (i = offsets[0]; i < msizes[0]; i++) {
          e1d[i] = temp;
          c1d[i] = coef *(1.0 + coef_epsilon * e1d[i]);
      }
    }
    else {
      for (i = offsets[0]; i < msizes[0]; i++) {
          c1d[i] = coef *(1.0 + coef_epsilon * e1d[i]);
      }
    }
}
else if (dim == 2) {
    if (simulation_time < 0.5 * dt) {
        for (j = offsets[1]; j < msizes[1]; j++) {
            for (i = offsets[0]; i < msizes[0]; i++) {
                e2d[j][i] = temp;
                c2d[j][i] = coef *(1.0 + coef_epsilon * e2d[j][i]);
            }
        }
    }
    else {
        for (j = offsets[1]; j < msizes[1]; j++) {
            for (i = offsets[0]; i < msizes[0]; i++) {
                c2d[j][i] = coef *(1.0 + coef_epsilon * e2d[j][i]);
            }
        }
    }
}
else if (dim == 3) {
```

```
        if (simulation_time < 0.5 * dt) {
            for (k = offsets[2]; k < msizes[2]; k++) {
                for (j = offsets[1]; j < msizes[1]; j++) {
                    for (i = offsets[0]; i < msizes[0]; i++) {
                        e3d[k][j][i] = temp;
                        c3d[k][j][i] = coef *(1.0 + coef_epsilon * e3d[k][j][i]);
                    }
                }
            }
        }
        else {
            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 *(1.0 + coef_epsilon * e3d[k][j][i]);
                    }
                }
            }
        }
    }
}
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)
{
    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)
{
  int i0, i, d;
  int *sizes;
  double width, hl, tmp, t_add, *coordl, *coordr, ctr[3], dc[3], x;
  double *e1d;

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

  sizes  = m->sizes;
  coordl = m->coordl;
```

```
    coordr = m->coordr;

    if (ebdry[0] != ebdry[1]) {
        printf("ERROR: ebdry[0] != ebdry[1]\n");
        exit(0);
    }
    width = 0.25 * (coordr[0] - coordl[0]);
    hl    = 0.5 * (coordr[0] - coordl[0]);
    tmp = hl/width;
    t_add  = ebdry[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];
    }
    e1d = m->e1d;
    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;
        e1d[i] = t_add + t_initial * exp(-tmp * tmp);
    }
    return 0;
}



int set_initial_temp(H_Mesh *m)
{
    int i, j, k;
    int *sizes;
    double *e1d, **e2d, ***e3d;
    double *sh1d, **sh2d, ***sh3d, *sn1d, **sn2d, ***sn3d;
    H_Mesh *child;

    sizes = m->sizes;
    e1d = m->e1d;
```

```
    e2d = m->e2d;
    e3d = m->e3d;
    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++) {
//            e1d[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++) {
//                e2d[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++) {
//                    e3d[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;
        e1d = child->e1d;
        e2d = child->e2d;
        e3d = child->e3d;

        if (dim == 1) {
            for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
                e1d[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++) {
                    e2d[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++) {
                        e3d[k][j][i] = 0.0;
                    }
                }
            }
        }
    }
    return 0;
}

int set_mesh(double *coordl, double *coordr, int *sizes, H_Mesh *m)
{
    double **e2d, **en2d, **eh2d, **c2d, **cx2d, **cy2d;
    double ***e3d, ***en3d, ***eh3d, ***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->e1d  = (double *) malloc(lsize * sizeof(double));
   m->en1d = (double *) malloc(lsize * sizeof(double));
   m->eh1d = (double *) malloc(lsize * sizeof(double));
   m->en1d = (double *) malloc(lsize * sizeof(double));
   m->eh1d = (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, &e2d);
   set_2d_form(sizes, nbdy, &eh2d);
   set_2d_form(sizes, nbdy, &en2d);
   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->e2d  = e2d;
    m->eh2d = eh2d;
    m->en2d = en2d;
    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, &e3d);
    set_3d_form(sizes, nbdy, &eh3d);
    set_3d_form(sizes, nbdy, &en3d);
    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->e3d  = e3d;
    m->eh3d = eh3d;
    m->en3d = en3d;
    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)
{
  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->e1d = NULL;
    m->e2d = NULL;
    m->e3d = 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)
{
  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)
{
    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)
{
    char name[125];
    char mesh_name[] = "mesh";
    char var_name[]  = "E";
    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, *e1d, **e2d, ***e3d;
    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) {
      e1d = m->e1d;
      e2d = m->e2d;
      e3d = m->e3d;
    }
    else {
      e1d = m->en1d;
      e2d = m->en2d;
      e3d = m->en3d;
```

```
}
sizes = m->sizes;
coordl = m->coordl;
coordr = m->coordr;
for (d = 0; d < dim; d++) {
   dc[d] = (coordr[d] - coordl[d])/(double) sizes[d];
}
printf("dump file at t = %e\n", t);

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], ebdry[0], coordr[0] + 0.5*dc[0], ebdry[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, e1d[i], c1d[i]);
   }
   fclose(fp);
}
else {
   sprintf(name, "data_%05d", idump);
   printf("dump file %s at t = %e\n", name, t);
   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) {
    for (j = 0; j < sizes[1] + nbdy + nbdy; j++) {
        for (i = 0; i < sizes[0] + nbdy + nbdy; i++) {
            if (e2d[j][i] < 1.0) e2d[j][i] = 1.0;
        }
    }
    set_radiative_coef(m);
    ii = 0;
    for (j0 = 0; j0 < sizes[1]; j0++) {
        j = nbdy + j0;
        for (i0 = 0; i0 < sizes[0]; i0++) {
            i = nbdy + i0;
            v[ii] = e2d[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] = e3d[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;
}
```