The White Paper on ELLAM Implementation in C++

Jiangguo (James) Liu [1]

March 11, 2009

[1]Department of Mathematics, Colorado State University, Fort Collins, CO 80523-1874, USA, liu@math.colostate.edu

Contents

List of Tables

List of Figures

Chapter 1

ELLAM for IBVP of Convection-Diffusion Equations

1.1 Problem

Let Ω be a (not necessarily rectangular) domain in $\mathbb{R}^d$ ($d = 1, 2,$ or 3) and $[0, T]$ be a time period. We consider the following linear convection-diffusion equation:

$$(Ku)_t + \nabla \cdot (\mathbf{V}u - \mathbf{D}\nabla u) + Ru = f(\mathbf{x}, t), \qquad \mathbf{x} \in \Omega,\ t \in (0, T], \tag{1.1}$$

where where

$K(\mathbf{x}, t)$	retardation
$\mathbf{V}(\mathbf{x}, t)$	velocity
$\mathbf{D}(\mathbf{x}, t)$	diffusion
$K(\mathbf{x}, t)$	reaction
$f(\mathbf{x}, t)$	source/sink

$u(\mathbf{x}, t)$ is the unknown function, $K(\mathbf{x}, t)$ is a retardation coefficient, $\mathbf{V}(\mathbf{x}, t)$ is a fluid velocity field, $\mathbf{D}(\mathbf{x}, t)$ is a diffusion-dispersion tensor, and $f(\mathbf{x}, t)$ is a source term. It is assumed that $K(\mathbf{x}, t)$ has positive lower and upper bounds. In addition, $\mathbf{D}(\mathbf{x}, t)$ is taken as a real symmetric positive definite matrix with uniform lower and upper bounds for its eigenvalues that are independent of $(\mathbf{x}, t)$.

An initial condition

$$u(\mathbf{x}, 0) = u_0(\mathbf{x}), \quad \mathbf{x} \in \Omega \tag{1.2}$$

is posed as usual. But boundary conditions needs more attention.

Let $\partial\Omega$ be the boundary of Ω, then we can decompose the space-time boundary $\Gamma := \partial\Omega \times [0,T]$ as $\Gamma = \Gamma^I \cup \Gamma^O \cup \Gamma^N$ where

$$\begin{aligned} \Gamma^I &:= \{(\mathbf{x},t) \mid \mathbf{x} \in \partial\Omega, t \in [0,T], \mathbf{V}(\mathbf{x},t)\cdot\mathbf{n}(\mathbf{x}) < 0\} \\ \Gamma^O &:= \{(\mathbf{x},t) \mid \mathbf{x} \in \partial\Omega, t \in [0,T], \mathbf{V}(\mathbf{x},t)\cdot\mathbf{n}(\mathbf{x}) > 0\} \\ \Gamma^N &:= \{(\mathbf{x},t) \mid \mathbf{x} \in \partial\Omega, t \in [0,T], \mathbf{V}(\mathbf{x},t)\cdot\mathbf{n}(\mathbf{x}) = 0\} \end{aligned} \tag{1.3}$$

are the inflow, outflow, and noflow boundaries, respectively, and $\mathbf{n}(\mathbf{x})$ is the outward unit normal vector.

In general, an inflow boundary during one time period might becom an outflow or a noflow boundary in the next time period or vice versa.

On the noflow boundary, we have

$$(\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n} \equiv 0. \tag{1.4}$$

On the inflow or ouflow boundary, one can mathematically speicify Dirichlet, Neumann, or Robin (total flux) condition:

$$\begin{aligned} u(\mathbf{x},t) &= g_1^{type}(\mathbf{x},t), \quad (\mathbf{x},t) \in \Gamma^{type}, \quad \textit{Dirichlet} \\ -\mathbf{D}\nabla u(\mathbf{x},t)\cdot\mathbf{n} &= g_2^{type}(\mathbf{x},t), \quad (\mathbf{x},t) \in \Gamma^{type}, \quad \textit{Neumann} \\ (\mathbf{V}u - \mathbf{D}\nabla u)(\mathbf{x},t)\cdot\mathbf{n} &= g_3^{type}(\mathbf{x},t), \quad (\mathbf{x},t) \in \Gamma^{type}, \quad \textit{Robin} \end{aligned} \tag{1.5}$$

where $type = I$ or O represents the inflow or outflow boundary type. In applications, however, an inflow Dirichlet or Robin condition and an outflow Dirichlet or Neumann condition are often posed. For example, common boundary conditions in petroleum reservior simulation are noflow boundary conditions. For groundwater contamination simulation, a Dirichlet or Robin condition is usually specified at the inflow boundary while a Dirichlet or Neumann boundary condition is posed at the outflow boundary.

1.2 Time Discretization and Localized Test Functions

Let $0 = t_0 < t_1 < \cdots < t_{n-1} < t_n < \cdots < t_N = T$ be a partition of $[0,T]$ with $\Delta t_n := t_n - t_{n-1}$.

We choose test functions $w(\mathbf{x},t) \in H^1(\Omega \times [0,T])$ in such a way that they vanish outside the space-time strip $\Omega \times (t_{n-1}, t_n]$ and are discontinuous in time at time t_{n-1}. Then integration (in time) by parts, or to be accurate, Green's formula, leads us to the following weak formulation:

Find $u(\mathbf{x},t) \in H^1(\Omega \times (0,T])$ so that for any test function $w(\mathbf{x},t)$, the following equation holds

$$\begin{aligned} &\int_\Omega (Ruw)(\mathbf{x},t_n)d\mathbf{x} + \int_{J_n}\int_\Omega (\mathbf{D}\nabla u)\cdot\nabla w \; d\mathbf{x}dt \\ &\quad + \int_{\Gamma_n} (\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\; w \; d\mathbf{y}dt - \int_{\Sigma_n} u(w_t + \mathbf{V}\cdot\nabla w)\; d\mathbf{x}dt \\ &\int_\Omega (Ruw)(\mathbf{x},t_{n-1}^+)d\mathbf{x} + \int_{\Sigma_n} f(\mathbf{x},t)w(\mathbf{x},t)d\mathbf{x}dt, \end{aligned} \tag{1.6}$$

where $\Sigma_n := \Omega \times J_n$ and $d\mathbf{y}dt$ is the differential element on the space-time boundary $\partial\Omega \times J_n$ and

$$w(\mathbf{x},t_{n-1}^+) := \lim_{t\to t_{n-1}^+} w(\mathbf{x},t) \tag{1.7}$$

takes into account the fact that $w(\mathbf{x},t)$ is discontinuous in time at time t_{n-1}.

1.3 Characteristics

We define the characteristic $\mathbf{y}(s;\mathbf{x},t)$ passing through $(\mathbf{x},t), t \in J_n$ by the following initial value problem of the ordinary differential equation:

$$\begin{cases} \dfrac{d\mathbf{y}}{ds} = \dfrac{\mathbf{V}(\mathbf{y},s)}{K(\mathbf{y},s)} \\ \mathbf{y}(s;\mathbf{x},t)|_{s=t} = \mathbf{x} \end{cases} \tag{1.8}$$

The notation $\mathbf{y}(s;\mathbf{x},t)$ can refer to tracking either forward or backward. In particular, we define

$$\mathbf{x}^* = \mathbf{y}(t_{n-1};\mathbf{x},t_n), \tag{1.9}$$

$$\tilde{\mathbf{x}} = \mathbf{y}(t_n;\mathbf{x},t_{n-1}). \tag{1.10}$$

In other words, $(\mathbf{x},t_n)$ backtracks to $(\mathbf{x}^*,t_{n-1})$ while $(\mathbf{x},t_{n-1})$ tracks forward to $(\tilde{\mathbf{x}},t_n)$.

In numerical schemes, an exact tracking of characteristics is preferred whenever possible. If exact tracking is impossible, Euler quadrature or Runge-Kutta quadrature with multiple micro time steps within a global time step are often used.

Note that in many applications, (1.1) is usually coupled with an potential or pressure equation whose solution is often obtained via the mixed finite element method. For example, a Raviart-Thomas space is often used for the velocity field, which is calculated on the interfaces of each cell. With each cell, one component is piecewise linear in x and piecewise constant in y, while the other component is piecewise linear in y and piecewise constant in x.

1.4 Adjoint Equation

To eliminate the last term on the left-hand side of the variational form, we require the test functions satisfy the adjoint equation

$$w_t + \mathbf{V} \cdot \nabla w = 0. \tag{1.11}$$

Along characteristics the adjoint equation becomes an ordinary differential equation

$$\begin{cases} \dfrac{d}{ds} w(\mathbf{y}(s;\mathbf{x},t),s) = 0, \\ w(\mathbf{y}(s;\mathbf{x},t),s)|_{s=t} = w(\mathbf{x},t), \end{cases}$$

which implies that test functions are constants along characteristics.

Choosing $(\mathbf{x},t) = (\mathbf{x},t_n)$, we see that once the test functions $w(\mathbf{x},t_n)$ are specified at time t_n, they are completely determined in the space-time strip $\Omega \times J_n$. Therefore, we obtain

$$\begin{aligned} &\int_\Omega u(\mathbf{x},t_n) w(\mathbf{x},t_n) d\mathbf{x} + \int_{J_n}\int_\Omega (\mathbf{D}\nabla u)\cdot\nabla w \; d\mathbf{x}dt \\ &\quad + \int_{J_n}\int_{\partial\Omega} (\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\; w \; d\mathbf{y}dt \\ &= \int_\Omega u(\mathbf{x},t_{n-1}) w(\mathbf{x},t_{n-1}^+) d\mathbf{x} + \int_{J_n}\int_\Omega f(\mathbf{x},t) w(\mathbf{x},t) d\mathbf{x}dt, \end{aligned}$$

where the second term on the left-hand side is called diffusion term, the last term on the left-hand side is boundary term while last term on the right-hand side is source term.

1.5 Boundaries, Flows, and Tracking

Let $\Gamma_n := \partial\Omega \times J_n$ and $\Gamma_n^I, \Gamma_n^O, \Gamma_n^N$ be the inflow, outflow, and noflow space-time boundaries during time period J_n defined by

$$\begin{aligned} \Gamma_n^I &:= \{(\mathbf{x},t) \mid \mathbf{x} \in \partial\Omega, t \in J_n, \mathbf{V}(\mathbf{x},t)\cdot\mathbf{n}(\mathbf{x}) < 0\} \\ \Gamma_n^O &:= \{(\mathbf{x},t) \mid \mathbf{x} \in \partial\Omega, t \in J_n, \mathbf{V}(\mathbf{x},t)\cdot\mathbf{n}(\mathbf{x}) > 0\} \\ \Gamma_n^N &:= \{(\mathbf{x},t) \mid \mathbf{x} \in \partial\Omega, t \in J_n, \mathbf{V}(\mathbf{x},t)\cdot\mathbf{n}(\mathbf{x}) = 0\} \end{aligned} \tag{1.12}$$

To accurately measure the time period taken for a particle to move along a characteristic from the previous time step or the inflow boundary to the current time step or the outflow boundary, we introduce space-time (location-moment) dependent time steps.

For any $\mathbf{x} \in \Omega$ and $t = t_n$, if the characteristic backtracks to a point within Ω at time t_{n-1}, then we define

$$\Delta t^I(\mathbf{x}, t_n) = t_n - t_{n-1}. \tag{1.13}$$

If the characteristic backtracks to a point on the inflow boundary for some $t^* \in (t_{n-1}, t_n]$, then we define

$$\Delta t^I(\mathbf{x}, t_n) = t_n - t^*. \tag{1.14}$$

Similarly, for any $(\mathbf{y}, t) \in \Gamma_n^O$, we define

$$\Delta t^O(\mathbf{y}, t) = t - t_{n-1} \tag{1.15}$$

if the characteristic backtracks to within Ω at time t_{n-1} or

$$\Delta t^O(\mathbf{y}, t) = t - t^* \tag{1.16}$$

if the characteristic backtracks to the inflow boundary at time $t^* \in (t_{n-1}, t)$.

Figure 1.1 might be helpful for understanding the above ideas. Assume that the left and front faces of the cube are inflow boundaries during time period J_n and the right and back faces are outflow space-time boundaries. Of course, Ω at time t_{n-1} and Ω at time t_n can be viewed as inflow and outflow boundaries, respectively. Furthermore, suppose A_1, A_2 are in Ω at time t_{n-1}, B_1, B_2 are in Ω at time t_n, C_1, C_2 are on the inflow boundary Γ_n^I, and D_1, D_2 are on the outflow boundary Γ_n^O. So we use Δt^I to measure streamlines A_1B_1, C_1B_2 (of course $\Delta t^I = \Delta t_n$ for A_1B_1) and Δt^O to measure streamlines A_2D_1, C_2D_2.

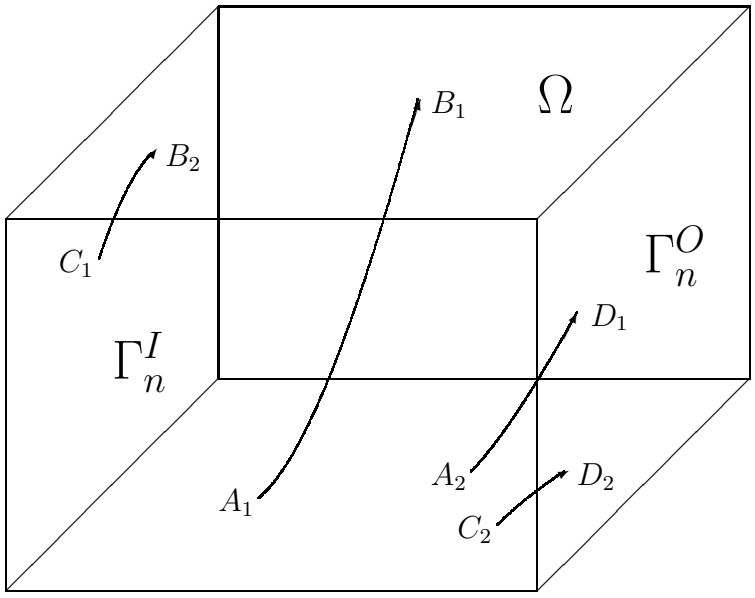


Figure 1.1: *Flows, characteristics, and boundaries*

1.6 Approximations of Diffusion and Source Terms

Let $\Sigma_n := \Omega \times J_n$ and Σ_n^O be the set of points in Σ_n that will flow out through Γ_n^O during time period J_n. By enforcing backward Euler quadrature at the current time t_n and on the outflow space-time boundary Γ_n^O, we get

$$\begin{aligned}
&\int_{\Sigma_n} f(\mathbf{x},t)w(\mathbf{x},t)d\mathbf{x}dt \\
&= \int_{\Sigma_n \setminus \Sigma_n^O} f(\mathbf{x},t)w(\mathbf{x},t)d\mathbf{x}dt + \int_{\Sigma_n^O} f(\mathbf{x},t)w(\mathbf{x},t)d\mathbf{x}dt \\
&= \int_{\Omega} \Delta t^I(\mathbf{x},t_n) f(\mathbf{x},t_n)w(\mathbf{x},t_n)d\mathbf{x} \\
&\quad + \int_{\Gamma_n^O} \Delta t^O(\mathbf{y},t) f(\mathbf{y},t)w(\mathbf{y},t)\ (\mathbf{V}\cdot\mathbf{n})d\mathbf{y}dt \\
&\quad + E(f,w). \qquad (1.17)
\end{aligned}$$

Similarly, the diffusion term can be approximated as follows

$$\begin{aligned}
&\int_{\Sigma_n} ((\mathbf{D}\nabla u)\cdot\nabla w)(\mathbf{x},t)d\mathbf{x}dt \\
&= \int_{\Omega} \Delta t^I(\mathbf{x},t_n)((\mathbf{D}\nabla u)\cdot\nabla w)(\mathbf{x},t_n)d\mathbf{x} \\
&\quad + \int_{\Gamma_n^O} \Delta t^O(\mathbf{y},t)((\mathbf{D}\nabla u)\cdot\nabla w)(\mathbf{y},t)\ (\mathbf{V}\cdot\mathbf{n})d\mathbf{y}dt \\
&\quad + E(D,u,w). \qquad (1.18)
\end{aligned}$$

1.7 Decomposition of Boundary Term

Notice that on noflow boundary, the total flux vanishes,

$$(\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n} \equiv 0, \qquad (1.19)$$

so the boundary term can be further decomposed into two parts:

$$\begin{aligned}
&\int_{\Gamma_n} (\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\ w(\mathbf{y},t)d\mathbf{y}dt \\
&= \int_{\Gamma_n^I} (\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\ w(\mathbf{y},t)d\mathbf{y}dt \qquad (1.20) \\
&\quad + \int_{\Gamma_n^O} (\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\ w(\mathbf{y},t)d\mathbf{y}dt.
\end{aligned}$$

More details on handling these two boundary terms will be provided later.

1.8 Reference Equation

Dropping the error terms in the source and diffusion terms, we finally obtain the following reference equation:

Find $u(\mathbf{x},t) \in H^1(\Omega \times (t_{n-1},t_n])$ so that the following holds for any $w(\mathbf{x},t) \in H^1(\Omega \times (t_{n-1},t_n])$

$$
\begin{aligned}
&\int_\Omega u(\mathbf{x},t_n)w(\mathbf{x},t_n)d\mathbf{x} \\
&\quad + \int_\Omega \Delta t^I(\mathbf{x},t_n)((\mathbf{D}\nabla u)\cdot\nabla w)(\mathbf{x},t_n)d\mathbf{x} \\
&\quad + \int_{\Gamma_n^O} \Delta t^O(\mathbf{y},t)((\mathbf{D}\nabla u)\cdot\nabla w)\ (\mathbf{V}\cdot\mathbf{n})d\mathbf{y}dt \\
&\quad + \int_{\Gamma_n^O} (\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\ w(\mathbf{y},t)d\mathbf{y}dt \\
&= \int_\Omega u(\mathbf{x}^*,t_{n-1})w(\mathbf{x},t_n)\mathbf{J}(\mathbf{x}^*,\mathbf{x})\ d\mathbf{x} \\
&\quad + \int_\Omega \Delta t^I(\mathbf{x},t_n)f(\mathbf{x},t_n)w(\mathbf{x},t_n)d\mathbf{x} \\
&\quad + \int_{\Gamma_n^O} \Delta t^O(\mathbf{y},t)f(\mathbf{y},t)w(\mathbf{y},t)\ (\mathbf{V}\cdot\mathbf{n})d\mathbf{y}dt \\
&\quad - \int_{\Gamma_n^I} (\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\ w(\mathbf{y},t)d\mathbf{y}dt,
\end{aligned}
\tag{1.21}
$$

where $\mathbf{x}^* := \mathbf{y}(t_{n-1};\mathbf{x},t_n)$ and $\mathbf{J}(\mathbf{x}^*,\mathbf{x})$ is the Jacobian.

Chapter 2

Finite Element Methods within ELLAM Framework

One advantage of the ELLAM formulation is that there is basically no restriction on spatial discretizations. This leaves the door open for finite element methods and other methods, e.g., wavelet methods . For now, we choose trial functions and test fucntions from the standard finite element methods to develop numerical schemes for boundary initial value problems.

The main advantage and also the major difficulty of the ELLAM method is characteristic tracking. Actually there is no any restriction on the spatial domain in the ELLAM formulation. The main difficulty lies in locating the foot or head of a characteristic. Clearly, rectangular meshes offer a great deal of convenience for the locating mentioned above. So from now on we shall focus rectangular domains and hence rectangular meshes.

2.1 Partitions of Spatial Domain and Outflow Space-Time Boundary and Courant Number

Note that in the ELLAM formulation, the unknown function is defined on both Ω at time t_n and the outflow space-time boundary Γ_n^O. This is in contrast to most numerical schemes for time-dependent problems, in which the unknown is typically defined only on the spatial domain Ω at time t_n.

We first define a quasi-uniform spatial partition $\mathcal{T}_h$ of mesh size h for domain Ω at time t_n, then extend it to Γ_n^O.

During time period J_n, some of the mass on domain Ω at time t_{n-1} will flow out through the outflow space-time boundary Γ_n^O. The number of spatial degrees of freedom crossing Γ_n^O is essentially the Courant number in the normal direction. To preserve this information, one should partition Γ_n^O in time direction with about the same number of subintervals. In other words, the local time partition on Γ_n^O should satisfy the CFL condition.

The ELLAM formulation uses Lagrangian coordinates and thus is not subject to the CFL restriction. However, on the outflow space-time boundary Γ_n^O, the temporal discretization should obey the CFL condition for the reason of accuracy and stability.

To be specific, we define Courant number on the outflow space-time boundary Γ_n^O by

$$Cr := \max_{(\mathbf{y},t)\in\Gamma_n^O} |\mathbf{V}(\mathbf{y},t)\cdot\mathbf{n}(\mathbf{y})|\frac{\Delta t_n}{h} \tag{2.1}$$

and NC as the ceiling of Cr. Then we define a uniform temporal partition on Γ_n^O by

$$t_{n,k} := t_n - k\frac{\Delta t_n}{NC}, \quad k = 0, 1, \cdots, NC. \tag{2.2}$$

Finally, we extend the spatial partition $\mathcal{T}_h$ on Ω at time t_n to Γ_n^O with local time refinement given by (2.2). The united partition will be denoted by $\mathcal{T}_{h,\Delta t}$.

2.2 Two Groups of Trial and Test Functions

Note that in the reference equation (1.21), not all terms appear simultaneously. The first two terms on the left-hand side and the second term on the right-hand side correspond to only the test functions defined on Ω at time t_n, the third and fourth terms on the left-hand side and the third term on the right-hand side correspond to only the test functions defined on the outflow space-time boundary Γ_n^O, while the last term on the left-hand side and the first term on the right-hand side correspond to possibly both.

So there are two groups of terms and equations. One group is related to the trial and test functions defined on Ω at time t_n, the other corresponds to the trial and test functions defined on the outflow space-time boundary Γ_n^O.

These two groups of equations might be coupled or decoupled, depending on proposed boundary conditions.

2.3 Interior Nodes and Noflow Boundary

For any test function w, let supp(w) be its support on domain Ω or the outflow boundary Γ_n^O, and Σ_w^* be the prism obtained by backtracking supp(w) along characteristics from Ω at time t_n or Γ_n^O to Ω at time t_{n-1} or the inflow boundary Γ_n^I.

If the test function w is defined on Ω at time t_n and Σ_w^* does not intersect the inflow boundary Γ_n^I during time period J_n, then we obtain the following equations:

$$\begin{aligned} &\int_\Omega U(\mathbf{x},t_n)w(\mathbf{x},t_n)d\mathbf{x} \\ &\qquad + \int_\Omega \Delta t\ ((\mathbf{D}\nabla U)\cdot\nabla w)(\mathbf{x},t_n)d\mathbf{x} \\ = &\int_\Omega U(\mathbf{x},t_{n-1})w(\mathbf{x},t_{n-1}^+)d\mathbf{x} \\ &\qquad + \int_\Omega \Delta t\ f(\mathbf{x},t_n)w(\mathbf{x},t_n)d\mathbf{x}. \end{aligned} \tag{2.3}$$

This implies that if all boundaries are noflow or all nodes are interior nodes, then the coefficient matrix of the whole system is 9-banded, symmetric and positive definite.

2.4 Inflow Boundary Conditions

No matter the test function is defined on Ω at time t_n or on the outflow space-time boundary Γ_n^O, we have to deal with the inflow boundary term

$$\int_{\Gamma_n^I} (\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\ w(\mathbf{y},t)\ dS \tag{2.4}$$

once Σ_w^* intersects Γ_n^I.

For a **inflow Robin (total flux) condition**, it becomes

$$\int_{\Gamma_n^I} (\mathbf{V}u - \mathbf{D}\nabla u) \cdot \mathbf{n}\, w(\mathbf{y},t)\, dS = -\int_{\Gamma_n^I} g_3^I w(x,y,t) d\mathbf{y} dt \tag{2.5}$$

and can be moved to the right-hand side of the reference equation.

When we backtrack the support of a test function, its image is splitted into two parts, one in Ω at time t_{n-1} and the other on the inflow space-time boundary Γ_n^I. Note that the factor $\Delta t^I(\mathbf{x}, t_n)$ in the second term or $\Delta t^I(\mathbf{y}, t_n)$ in the third term of the left-hand side is smaller than Δt_n. Nevertheless, the system still has a 9-banded, symmetric positive definite coefficient matrix.

For a **inflow Dirichlet condition**, the function values of u are known on the inflow space-time boundary Γ_n^I, and so are the tangential derivatives. But its normal derivatives are unknown on Γ_n^I. This means the unknown boundary diffusive flux is coupled with unknown interior function values. To resolve the problem, we approximate $\nabla U(\mathbf{y}, t)$ on the inflow boundary Γ_n^I implicitly by $\nabla U(\mathbf{x}(t_n; \mathbf{y}, t), t_n)$. This removes the difficulty of evaluating an unknown diffusive boundary flux. The error is small since it is along characteristics. Be aware that this term introduces **nonsymmetry** to the coeffcient matrix near the inflow boundary.

For a **inflow Neumann condition**, readers are referred to [6].

2.5 Outflow Boundary Conditions

Outflow boundary conditions involve the third term

$$\int_{\Gamma_n^O} \Delta t^O(\mathbf{y}, t)((\mathbf{D}\nabla u) \cdot \nabla w)(\mathbf{y}, t)\, (\mathbf{V} \cdot \mathbf{n}) dS \tag{2.6}$$

and the fourth term

$$\int_{\Gamma_n^O} (\mathbf{V}u - \mathbf{D}\nabla u) \cdot \mathbf{n}\, w(\mathbf{y}, t) dS \tag{2.7}$$

on the left-hand side of the reference equation.

For a **outflow Dirichlet condition**, the function u and hence its tangential derivatives are known on the outflow space-time boundary Γ_n^O. Only the normal derivatives of the trial fucntion U are taken as unknowns. But

their equations are decoupled from those associated with the values of the trial function U at interior nodes and are needed only for mass conservation. Therefore we omitt them here.

For a **outflow Neumann condition** or a **Outflow Robin condition**, some techniques have been successfully developed in [6].

2.6 FEM Assembly

FEM assembly

$$
\begin{aligned}
&\sum_E \int_E \Big[K(\mathbf{x},t_n)u(\mathbf{x},t_n)w(\mathbf{x},t_n) + \Delta t^I(\mathbf{x},t_n)((\mathbf{D}\nabla u)\cdot\nabla w)(\mathbf{x},t_n)\Big]d\mathbf{x} \\
&\quad + \int_{\Gamma_n^O} \Big[\Delta t^O(\mathbf{y},t)((\mathbf{D}\nabla u)\cdot\nabla w)\ (\mathbf{V}\cdot\mathbf{n}) \\
&\qquad +(\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\ w(\mathbf{y},t)\Big]d\mathbf{y}dt \\
&= \sum_E \int_E \Big[K(\mathbf{x},t_{n-1})u(\mathbf{x}^*,t_{n-1})w(\mathbf{x},t_n)\mathbf{J}(\mathbf{x}^*,\mathbf{x}) \\
&\qquad +\Delta t^I(\mathbf{x},t_n)f(\mathbf{x},t_n)w(\mathbf{x},t_n)\Big]d\mathbf{x} \\
&\quad + \int_{\Gamma_n^O} \Delta t^O(\mathbf{y},t)f(\mathbf{y},t)w(\mathbf{y},t)\ (\mathbf{V}\cdot\mathbf{n})d\mathbf{y}dt \\
&\quad - \int_{\Gamma_n^I} (\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\ w(\mathbf{y},t)d\mathbf{y}dt,
\end{aligned}
\tag{2.8}
$$

where $\mathbf{x}^* := \mathbf{y}(t_{n-1};\mathbf{x},t_n)$ and $\mathbf{J}(\mathbf{x}^*,\mathbf{x})$ is the Jacobian.

Chapter 3

Implementation

3.1 Assumptions on Problem and Method

Our implementation makes the following assumptions:

1. We consider a single-phase linear transport equation. If the velocity, diffusion, or reaction has mild nonlinearity, then the usual linearization should be OK.

2. The fluid velocity $\mathbf{V}(\mathbf{x}, t)$ is assumed to be known. It might be obtained from another program. As will be discussed later, the velocity field affects characteristic tracking to a great deal.

3. At present, we have not included the reaction term. When the reaction term is included, we shall use the points on characteritics to evaluate $\exp\left(-\int_{t_n}^{t^*} R(\mathbf{y}, s)ds\right)$. This requires certain accuracy in characteristic tracking and small global time steps.

4. We assume a rectangular domain Ω with a rectangular mesh parallel to the axes. Again this is mainly for convenience in characteristic tracking. A polygonal domain with a triangular mesh or even an unstructured mesh is not a hurdle to FEM assembly. Charatceristic tracking in such a mesh is fairly complicated but is now under our investigation.

5. We shall use Q_1 element on rectangles. Numerical experiments indicate that the h-version convergence suffices.

6. The current version is for 2-dim problems, but will be extended into a dimension-independent version in the near future.

7. We further assume the retardation, diffusion, and source terms are constants on each element, respectively. Therefore, compuations of the element mass and stiffness matrices and the source right-hand side will be greatly simplified. Usally elements are small and the above quantities do not experience big changes in a small element, so resulting errors should be small also.

8. Computing the mass contributon from the domain at the previous time step involves the Jacobian of characteristic tracking, which is of order $1 + \mathcal{O}(\Delta t)$. For simplicity, we shall use the Jacobian of the Euler method, even though Euler, Runge-Kutta, or other methods could be used for the actual tracking.

3.2 FEM Assembly

FEM assembly

$$
\begin{aligned}
&\sum_E \int_E \Big[K(\mathbf{x},t_n)u(\mathbf{x},t_n)w(\mathbf{x},t_n) + \Delta t^I(\mathbf{x},t_n)((\mathbf{D}\nabla u)\cdot\nabla w)(\mathbf{x},t_n)\Big]d\mathbf{x} \\
&\quad + \int_{\Gamma_n^O} \Big[\Delta t^O(\mathbf{y},t)((\mathbf{D}\nabla u)\cdot\nabla w)\ (\mathbf{V}\cdot\mathbf{n}) \\
&\qquad +(\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\ w(\mathbf{y},t)\Big]d\mathbf{y}dt \\
&= \sum_E \int_E \Big[K(\mathbf{x},t_{n-1})u(\mathbf{x}^*,t_{n-1})w(\mathbf{x},t_n)\mathbf{J}(\mathbf{x}^*,\mathbf{x}) \\
&\qquad +\Delta t^I(\mathbf{x},t_n)f(\mathbf{x},t_n)w(\mathbf{x},t_n)\Big]d\mathbf{x} \\
&\quad + \int_{\Gamma_n^O} \Delta t^O(\mathbf{y},t)f(\mathbf{y},t)w(\mathbf{y},t)\ (\mathbf{V}\cdot\mathbf{n})d\mathbf{y}dt \\
&\quad - \int_{\Gamma_n^I} (\mathbf{V}u - \mathbf{D}\nabla u)\cdot\mathbf{n}\ w(\mathbf{y},t)d\mathbf{y}dt,
\end{aligned}
\tag{3.1}
$$

where $\mathbf{x}^* := \mathbf{y}(t_{n-1};\mathbf{x},t_n)$ and $\mathbf{J}(\mathbf{x}^*,\mathbf{x})$ is the Jacobian.

3.3 Mesh and Timing

Actually, there is only one time step in the ELLAM methodology: from time t_{n-1} to time t_n. It assumes we have knowledge about u on Ω at t_{n-1} and on the inflow boundary $\partial\Omega$ during $[t_{n-1}, t_n]$, then solve for the solution on Ω at time t_n and the outflow boundary.

The domain Ω can be discretized into a rectangular mesh, or a triangular mesh, or an even mixed (unstructured) one, whatsoever makes sense. The mesh does not change during time steppping.

But we also need a partition for the outflow boundary Γ_n^O, which is a tensor product of the partiton on $\partial\Omega$ and a partition of $J_n = [t_{n-1}, t_n]$. As was discussed before, any partition of J_n should satisfy the CFL condition.

Now we have two group of unknowns on Ω_n and Γ_n^O, respectively, and two groups of trial/test functions and hence two groups of equations. The formulation of equations for each group is normal, but the assembly of these two groups needs some extra effort.

- Mesh for Ω;
- Mesh on Γ_n^O;
- Trial/test functions, equations for Ω_n;
- Trial/test functions, equations for Γ_n^O;
- Assembly of these two groups of equations

3.4 Numerical Integration

As was seen in the previous sections, ELLAM involves numerical integrations on finite elements. We can apply the trapzoid, Simpson, Gaussian 1-, 2-, 3-, 5- point quadrature on the standard interval $[-1, 1]$.

Suppose we have two standard quadratures $\mathrm{QuadX}, \mathrm{QuadY}$ on the interval $[-1, 1]$ and they are used for numerical integrations on the rectangular element with corners $(x_0, y_0), (x_1, y_1)$. The quadrature QuadX has N_x quadrature points $X_i, (0 \le i < N_x)$ and the quadrature QuadY has N_y

quadrature points $Y_j, (0 \leq j < N_y)$, respectively. Applying the following 2 mappings from $[x_0, x_1]$ or $[y_0, y_1]$ to $[-1, 1]$,

$$x = \frac{x_1 - x_0}{2} X + \frac{x_1 + x_0}{2}, \quad -1 \leq X \leq 1,$$
$$y = \frac{y_1 - y_0}{2} Y + \frac{y_1 + y_0}{2}, \quad -1 \leq Y \leq 1,$$

we can easily find the corresponding integration points in the intervals $[x_0, x_1]$ and $[y_0, y_1]$.

3.5 Matrices for the Reference Element

For a 2-dim rectangular mesh, all elements can be mapped back to the reference element: the unit square $[0, 1] \times [0, 1]$. Accordingly, their nodal basis functions and gradients can be expressed in terms of the nodal basis functions on the reference element. So let us work on the reference element first.

Table 3.1: Frequenetly used standard quadratures

name	**NumPts**	**point**	**weight**
trapzoid	2	-1.0	1.0
		1.0	1.0
Simpson	3	-1.0	0.333,333,333,333
		0.0	1.333,333,333,334
		1.0	0.333,333,333,333
Gaussian1	1	0.0	2
Gaussian2	2	-0.577,350,269,2??	1.0
		0.577,350,269,2??	1.0
Gaussian3	3	-0.774,596,669,2??	0.555,555,555,556
		0.0	0.888,888,888,889
		0.774,596,669,2??	0.555,555,555,556
Gaussian5	5	-0.906,179,845,939	0.236,926,885,056
		-0.538,469,310,106	0.478,628,670,499
		0.0	0.568,888,888,889
		0.538,469,310,106	0.478,628,670,499
		0.906,179,845,939	0.236,926,885,056

3.5.1 Mass Matrix for the Reference Element

Let $\Phi_m(X,Y), (m = 0,1,2,3)$ (or $00, 01, 10, 11$ in binary digits) be the $\mathcal{Q}_1$ nodal basis functions corresponding to the nodes $(0,0),(0,1),(1,0),(1,1)$ of the reference element. Then we have

$$\begin{aligned} \Phi_0 &= \Phi_{00} = (1-X)(1-Y), \\ \Phi_1 &= \Phi_{01} = (1-X)Y, \\ \Phi_2 &= \Phi_{10} = X(1-Y), \\ \Phi_3 &= \Phi_{11} = XY. \end{aligned} \tag{3.2}$$

These are the local trial and test functions on the reference element. The mass matrix for the reference element, which can be calculated exactly (without using any quadrature), is

$$M = [\langle \Phi_{m_1}, \Phi_{m_2} \rangle] = \begin{bmatrix} \frac{1}{9} & \frac{1}{18} & \frac{1}{18} & \frac{1}{36} \\ \frac{1}{18} & \frac{1}{9} & \frac{1}{36} & \frac{1}{18} \\ \frac{1}{18} & \frac{1}{36} & \frac{1}{9} & \frac{1}{18} \\ \frac{1}{36} & \frac{1}{18} & \frac{1}{18} & \frac{1}{9} \end{bmatrix} = \frac{1}{36} \begin{bmatrix} 4 & 2 & 2 & 1 \\ 2 & 4 & 1 & 2 \\ 2 & 1 & 4 & 2 \\ 1 & 2 & 2 & 4 \end{bmatrix} \tag{3.3}$$

All the functions are of product type, so we only need to evaluate integrals like

$$\int_0^1 s^2 ds = \int_0^1 (1-s)^2 ds = \frac{1}{3} \quad \text{and} \quad \int_0^1 s(1-s) ds = \frac{1}{6}.$$

3.5.2 Stiffness Matrix for the Reference Element

For the gradients, we have

$$\begin{aligned} \nabla\Phi_0 &= [-(1-Y), -(1-X)], \\ \nabla\Phi_1 &= [-Y, 1-X], \\ \nabla\Phi_2 &= [1-Y, -X], \\ \nabla\Phi_3 &= [Y, X]. \end{aligned} \tag{3.4}$$

Then the reference element stiffness matrix is

$$S = \left[\int_0^1 \int_0^1 \nabla\Phi_{m_1} \cdot \nabla\Phi_{m_2} \, dX dY \right] = \left[\text{ something } \right] \tag{3.5}$$

Note that we have

$$[(\Phi_0)_X, (\Phi_1)_X, (\Phi_2)_X, (\Phi_3)_X] = [-(1-Y), -Y, 1-Y, Y]$$

and

$$[(\Phi_0)_Y, (\Phi_1)_Y, (\Phi_2)_Y, (\Phi_3)_Y] = [-(1-X), 1-X, -X, X]\,.$$

So the stiffness matrix S can be split into 2 parts as $S = S_X + S_Y$ with

$$S_X = \begin{bmatrix} \frac{1}{3} & \frac{1}{6} & -\frac{1}{3} & -\frac{1}{6} \\ \frac{1}{6} & \frac{1}{3} & -\frac{1}{6} & -\frac{1}{3} \\ -\frac{1}{3} & -\frac{1}{6} & \frac{1}{3} & \frac{1}{6} \\ -\frac{1}{6} & -\frac{1}{3} & \frac{1}{6} & \frac{1}{3} \end{bmatrix} = \frac{1}{6} \begin{bmatrix} 2 & 1 & -2 & -1 \\ 1 & 2 & -1 & -2 \\ -2 & -1 & 2 & 1 \\ -1 & -2 & 1 & 2 \end{bmatrix} \tag{3.6}$$

and

$$S_Y = \frac{1}{6} \begin{bmatrix} 2 & -2 & 1 & -1 \\ -2 & 2 & -1 & 1 \\ 1 & -1 & 2 & -2 \\ -1 & 1 & -2 & 2 \end{bmatrix}. \tag{3.7}$$

3.5.3 Source RHS for the Reference Element

Assume the source term $f(x, y) \equiv 1$ on the reference element, then we have

$$\text{RHS} = \frac{1}{4} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \tag{3.8}$$

whence we compute integrals like

$$\int_0^1 s ds = \int_0^1 (1-s) ds = \frac{1}{4}.$$

3.6 Matrices on General Elements

Now we compute the mass and stiffness matrices on a general rectangular element with corner points (x_0, y_0) and (x_1, y_1). Let $\varphi_m(x, y), (m = 0, 1, 2, 3)$ (or $00, 01, 10, 11$ in binary digits) be the $\mathcal{Q}_1$ nodal basis functions corresponding to its nodes $(x_0, y_0), (x_0, y_1), (x_1, y_0), (x_1, y_1)$. Applying the following coordinate transforms,

$$\begin{aligned} x &= (x_1 - x_0)X + x_0, \quad 0 \le X \le 1, \\ y &= (y_1 - y_0)Y + y_0, \quad 0 \le Y \le 1, \end{aligned} \tag{3.9}$$

we get

$$\varphi_m(x, y) = \Phi_m(X, Y).$$

For convenience, let us denote $x_l = x_1 - x_0, y_l = y_1 - y_0$.

3.6.1 Mass Matrix

Based on the above changes of variables, we obtain

$$M = x_l y_l \frac{1}{36} \begin{bmatrix} 4 & 2 & 2 & 1 \\ 2 & 4 & 1 & 2 \\ 2 & 1 & 4 & 2 \\ 1 & 2 & 2 & 4 \end{bmatrix} \tag{3.10}$$

3.6.2 Stiffness Matrix

First we have

$$\frac{\partial}{\partial x}\varphi_m(x, y) = \frac{\partial \Phi_m(X, Y)}{\partial X}\frac{dX}{dx} = \frac{\partial \Phi_m(X, Y)}{\partial X}\frac{1}{x_l}.$$

Similarly,

$$\frac{\partial}{\partial y}\varphi_m(x, y) = \frac{\partial \Phi_m(X, Y)}{\partial Y}\frac{1}{y_l}.$$

Then we have

$$\nabla\varphi_{m_1} \cdot \nabla\varphi_{m_2} = \frac{1}{x_l^2}\frac{\partial \Phi_{m_1}}{\partial X}\frac{\partial \Phi_{m_2}}{\partial X} + \frac{1}{y_l^2}\frac{\partial \Phi_{m_1}}{\partial Y}\frac{\partial \Phi_{m_2}}{\partial Y}$$

Further calculations indicate that

$$\int_{x_0}^{x_1}\int_{y_0}^{y_1} \nabla\varphi_{m_1}\cdot\nabla\varphi_{m_2}dxdy = \frac{y_l}{x_l}\int_0^1\int_0^1 \frac{\partial\Phi_{m_1}}{\partial X}\frac{\partial\Phi_{m_2}}{\partial X}dXdY + \frac{x_l}{y_l}\int_0^1\int_0^1 \frac{\partial\Phi_{m_1}}{\partial Y}\frac{\partial\Phi_{m_2}}{\partial Y}dXdY$$

Therefore the stiffness matrix have to be split into 2 parts:

$$S = \frac{y_l}{x_l}S_X + \frac{x_l}{y_l}S_Y$$

that is,

$$S = \frac{y_l}{x_l}\frac{1}{6}\begin{bmatrix} 2 & 1 & -2 & -1 \\ 1 & 2 & -1 & -2 \\ -2 & -1 & 2 & 1 \\ -1 & -2 & 1 & 2 \end{bmatrix} + \frac{x_l}{y_l}\frac{1}{6}\begin{bmatrix} 2 & -2 & 1 & -1 \\ -2 & 2 & -1 & 1 \\ 1 & -1 & 2 & -2 \\ -1 & 1 & -2 & 2 \end{bmatrix}. \tag{3.11}$$

3.6.3 Source RHS

Similarly,

$$\text{RHS} = \frac{x_l y_l}{4}\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{3.12}$$

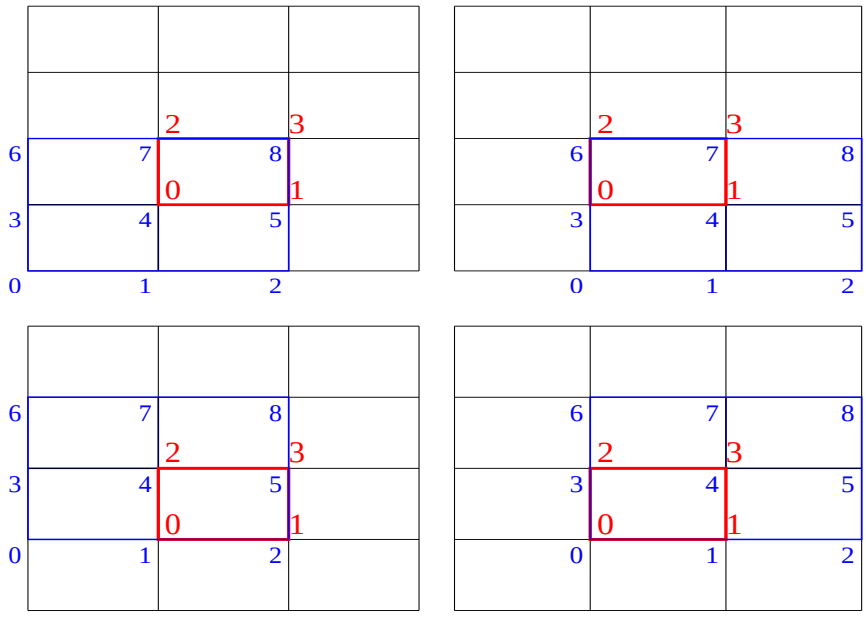
3.7 Assembly of Element Matrices and RHS

Actually, we also have the concept of **stencil** for FEM: a node and its neighboring nodes, as shown in the following table and figures.

Table 3.2: A typical rectangular element and its neighboring nodes

	0	1	2	3
0	4	5	7	8
1	3	4	6	7
2	1	2	4	5
3	0	1	3	4

Figure 3.1: A typical rectangular element and its neighboring nodes



3.8 Linear Solvers

When a linear system is symmetric positive, we employ the conjuagte gradient (CG) method as our linear solver, see [2] Algorithm 10.2.1 and flow chart (10.2.16). For the nonsymmetric case, we use GMRES instead.

In the following flow chart, ε is the (relative) tolerance and $k_{\max}$ is the iteration maximum, other notations are self-explanatory.

$k = 0$
$x =$ initial guess
$r = b - Ax$
$\gamma_0 = r^T r$
$\text{res} = \sqrt{\gamma_0}$
$\text{rhs} = \|b\|_2$
while $(k < k_{\max}) \ \wedge (\text{res} > \varepsilon \ \text{rhs})$
    $k = k + 1$
    if $k = 1$
        $p = r$
    else
        $\beta = \gamma_{k-1}/\gamma_{k-2}$
        $p = r + \beta p$
    end
    $q = Ap$
    $\delta = p^T q$
    $\alpha = \gamma_{k-1}/\delta$
    $x = x + \alpha p$
    $r = r - \alpha q$
    $\gamma_k = r^T r$
    $\text{res} = \sqrt{\gamma_k}$
end

3.9 Characteristic Tracking

Characteristic tracking is a vital part of ELLAM and also an important aspect for many numerical methods in computational fluid mechanics, as of my knowledge. Both forward and backward tracking are used in ELLAM. Althouh ELLAM is not subject to the CFL conditions in terms of stability, time steps should not be too large for the sake of accuracy. Optimal time steps for given data with minimal regularities are discussed in [4]. Based on these considerations, we require the temporal partition on the outflow boundary to satisfy the CFL conditions. For a typical step in the ELLAM framework, backtracking of characteristics is carried out from the domain Ω at time t_n or the outflow boundary Γ_n^O to the domain at t_{n-1} or the inflow boundary Γ_n^I. We shall use the same temporal partition on the outflow boundary for characteristic backtracking.

Mathematically, characteristic tracking is expressed as solving an initial value problem to an ODE

$$\begin{cases} \dfrac{d\mathbf{y}}{ds} = \mathbf{V}(\mathbf{y}, s) \\ \mathbf{y}(s; \mathbf{x}, t)|_{s=t} = \mathbf{x} \end{cases} \tag{3.13}$$

But it involves many aspects: a spatial domain Ω, a time period $[t_{n-1}, t_n]$, an outflow boundary Γ_n^O, a velocity vector field defined on the prism $\Sigma_n = \Omega \times [t_{n-1}, t_n]$, the proposed numerical methods (e.g. Euler or Runge-Kutta methods) and (not so obviously) meshes on the domain and the outflow boundary. If the mesh used in ELLAM is a triangular mesh instead a rectangular one, then some efforts have to be spent on locating the feet of the characteristics. In more general cases, the veclocity could be just a numerical solution produced by another numerical method and very likely defined on a different mesh of Ω. Then we have to further locate the intermediate node after each micro step in characteristic tracking. At this stage of design, however, we assume the mesh used in ELLAM is a rectangular mesh and the velocity is a function defined on $\Sigma_n = \Omega \times [t_{n-1}, t_n]$.

The code for implementing characteristic tracking should be dimension-independent. Staring from the current psotion/moment, we march to the next position/moment. The operations performed are a velocity vector times a time increment (a scalar) (which gives a position shift) and an addition of two positions (point ¡= point + vector*scalar).

Characteritics make a perfect case for a class. We name it **chara**, since **char** is already taken by most programming languages.

3.10 Examples

Example 0 The domain $\Omega = [-1, 1]^2$, the time period $[t_0, t_f] = [0, 1]$. The velocity $\mathbf{v} = (a, b)$, possibly $(1, 1)$, $(1, 0)$, or $(0, 1)$. The diffusion D is a small positive constant. For simplicity, we assume there is no reaction term: $R \equiv 0$. The exact solution is specified as

$$u(x, y, t) = \exp\left(-\frac{(x - x_c - at)^2 + (y - y_c - bt)^2}{2\sigma^2}\right).$$

This is a moving Gaussian hill. The center $(x_c + at, y_c + bt)$ should be well inside Ω so that u is numerically zero on $\partial\Omega$. An initial condition is derived accordingly.

Here are some details:

$$u_t = \frac{a(x - x_c - at) + b(y - y_c - bt)}{\sigma^2} u.$$

The velocity is div-free: $\nabla \cdot \mathbf{v} = 0$. Furthermore,

$$u_x = -\frac{x - x_c - at}{\sigma^2} u, \quad u_y = -\frac{y - y_c - bt}{\sigma^2} u,$$

and

$$\mathbf{v} \cdot \nabla u = -\frac{a(x - x_c - at) + b(y - y_c - bt)}{\sigma^2} u,$$

hence

$$u_t + \nabla \cdot (\mathbf{v}u) = 0.$$

Moreover,

$$\Delta u = -\frac{2}{\sigma^2} u + \frac{(x - x_c - at)^2 + (y - y_c - bt)^2}{\sigma^4} u.$$

Therefore,

$$f(x, y, t) = \frac{2D}{\sigma^2} u - D\frac{(x - x_c - at)^2 + (y - y_c - bt)^2}{\sigma^4} u.$$

In numerical runs, we set $D = 10^{-3}, \sigma = 0.0894$ so that $1/\sigma^2 = 62.5$.

Example 1 The domain $\Omega = [-1, 1]^2$. The velocity $\mathbf{v} = (-4y, 4x)$ is incompressible, i.e., $\nabla \cdot \mathbf{v} = 0$. The time period $[t_0, t_f] = [0, \pi/2]$, which is needed for one complete rotation. The characteristic with head (x, y, t) and foot $(x^*, y^*, 0)$ satisfies

$$\begin{bmatrix} x^* \\ y^* \end{bmatrix} = \begin{bmatrix} \cos(4t) & \sin(4t) \\ -\sin(4t) & \cos(4t) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \tag{3.14}$$

An initial condition

$$u_0(x, y) = \exp\Big(- \frac{(x - x_c)^2 + (y - y_c)^2}{2\sigma^2} \Big) \tag{3.15}$$

is a Gaussian hill centered at (x_c, y_c). The exact solution for the convection-diffusion equation with a constant diffusion $D > 0$ and $f = 0$ is

$$u(x, y, t) = \frac{2\sigma^2}{2\sigma^2 + 4Dt} \exp\Big(- \frac{(x^* - x_c)^2 + (y^* - y_c)^2}{2\sigma^2 + 4Dt} \Big), \tag{3.16}$$

where x^*, y^* are expressed in Equation (3.14). In our numerical experiment, we take $D = 10^{-4}$, $(x_c, y_c) = (-0.5, 0)$, and $\sigma = 0.0447$ so that $2\sigma^2 = 0.0040$. Then the maximum concentration at the final time t_f is 0.8642.

Chapter 4

Source Code of a Mini Version ELLAM

Here is the C++ code for a mini version ELLAM for the convection-diffusion equation on a uniform 2-dim rectangular mesh.

4.1 MiniELLAM: Head File

```
// MiniELLAM.h
// Jiangguo (James) Liu,  ColoState,  12/2008 - 02/2009

class ELLAM {
public:
  ELLAM();
  ~ELLAM();
  void setMesh(double a, double b, double c, double d, int nx, int ny);
  void setTime(double t0, double tf, int nt);
  void setDiffConst(double ExDiff);
  void setVelFxns(double(*ExVelx)(double x, double y, double t),
    double(*ExVely)(double x, double y, double t));
  void setSrsFxn(double(*ExSrs)(double x, double y, double t));
  void setVelData(double (*ExVx), double (*ExVy), int nx, int ny, int order);
  void setInitCond(double *U, int nx, int ny, int order);
  void getFinalSln(double *U, int nx, int ny, int order);
```

```
  void execute(int verbosity);
private:
  int NX, NY, NT;
  double XA, XB, YC, YD, T;
  double Deltax, Deltay, Deltat;
  double *X, *Y, *U0, *UT, *VX, *VY;
  double diff;
  double (*velx)(double x, double y, double t);
  double (*vely)(double x, double y, double t);
  double (*srs)(double x, double y, double t);
  void velxy(double &vx, double &vy, double x, double y, double t);
  void asmGlbMat(double *GlbMat, int *GlbMatNdx);
  void trknExEuler(double &xstar, double &ystar,
    double xn, double yn, double tn, double tn1);
  void asmGlbRHS4OldMass(double *GlbRHS, double *Un1, double tn);
  void asmGlbRHS4Srs(double *GlbRHS, double tn);
  void modGlbSysDirichletBndryCond(double *GlbMat, double *GlbRHS);
  int slvGlbSysBiCGStab(int n, double *x, double *A, int *M,
    double *b, double tol);
};
```

4.2 MiniELLAM: Source Code

```
// MiniELLAM.cpp
// Jiangguo (James) Liu,  ColoState,  12/2008 - 02/2009

#include <cmath>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include "MiniELLAM.h"
using namespace std;


// The 3rd order Gaussian quadrature on [-1,1]^2
```

```
static int GQR3npts = 9;
static double GQR3wab[9][3] = {
  0.07716049382716, -0.77459666924148, -0.77459666924148,
  0.12345679012346,  0.00000000000000, -0.77459666924148,
  0.07716049382716,  0.77459666924148, -0.77459666924148,
  0.12345679012346, -0.77459666924148,  0.00000000000000,
  0.19753086419753,  0.00000000000000,  0.00000000000000,
  0.12345679012346,  0.77459666924148,  0.00000000000000,
  0.07716049382716, -0.77459666924148,  0.77459666924148,
  0.12345679012346,  0.00000000000000,  0.77459666924148,
  0.07716049382716,  0.77459666924148,  0.77459666924148};


// ELLAM: default constructor

ELLAM::ELLAM()
{
  NX = 0;  NY = 0;  NT = 0;
  X = 0;  Y = 0;  U0 = 0;  UT = 0;  VX = 0;  VY = 0;  // Null pointers
  velx = 0;  vely = 0;  srs = 0;  // Null pointers to functions
}


// ELLAM: destructor

ELLAM::~ELLAM()
{
  delete[] X, Y, U0, UT, VX, VY;
}


// ELLAM: setup a uniform rectangular mesh on a given rectangle

void ELLAM::setMesh(double a, double b, double c, double d,
  int nx, int ny)
{
  int i, j;
```

```
  XA = a;  XB = b;  YC = c;  YD = d;
  NX = nx;  NY = ny;

  Deltax = (XB-XA)/NX;
  Deltay = (YD-YC)/NY;

  X = new double[NX+1];
  Y = new double[NY+1];

  X[0] = XA;
  for (i=1; i<NX; ++i)  X[i] = X[i-1] + Deltax;
  X[NX] = XB;

  Y[0] = YC;
  for (j=1; j<NY; ++j)  Y[j] = Y[j-1] + Deltay;
  Y[NY] = YD;

  return;
}


// ELLAM: setup a uniform time partition for marching

void ELLAM::setTime(double t0, double tf, int nt)
{
  T = tf - t0;  NT = nt;  Deltat = T/NT;
  return;
}


// ELLAM: set a positive diffision constant

void ELLAM::setDiffConst(double ExDiff)
{
  diff = ExDiff;
  return;
}
```

```
// ELLAM: set velocity data on the mesh

void ELLAM::setVelData(double (*ExVx), double (*ExVy),
  int nx, int ny, int order)
{
  int k;

  if (nx!=NX || ny!=NY) {
    cout << "Mesh mismatch!\n";
    exit(-1);
  }

  VX = new double[(NX+1)*(NY+1)];
  VY = new double[(NX+1)*(NY+1)];

  if (order==0) {
    for (k=0; k<(NX+1)*(NY+1); ++k) {
      VX[k] = ExVx[k];
      VY[k] = ExVy[k];
    }
  }

  return;
}


// ELLAM: set the velocity functions in the PDE

void ELLAM::setVelFxns(double(*ExVelx)(double x, double y, double t),
  double(*ExVely)(double x, double y, double t))
{
  velx = ExVelx;
  vely = ExVely;
  return;
}
```

```
// ELLAM: set the source function in the PDE

void ELLAM::setSrsFxn(double(*ExSrs)(double x, double y, double t))
{
  srs = ExSrs;
  return;
}


// ELLAM: interpolation velocity function
// Bilinear interpolation of the discrete velocity data
// Given x, y, assuming steady (independent of t)
// Return vx, vy (passing by reference)

void ELLAM::velxy(double &vx, double &vy, double x, double y, double t)
{
  int i = (int)floor((x-XA)/Deltax);
  int j = (int)floor((y-YC)/Deltay);

  double x0 = X[i];
  double x1 = X[i+1];
  double y0 = Y[j];
  double y1 = Y[j+1];

  double w00 = ((x1-x)/(x1-x0)) * ((y1-y)/(y1-y0));
  double w10 = ((x-x0)/(x1-x0)) * ((y1-y)/(y1-y0));
  double w01 = ((x1-x)/(x1-x0)) * ((y-y0)/(y1-y0));
  double w11 = ((x-x0)/(x1-x0)) * ((y-y0)/(y1-y0));

  double VX00 = VX[j*(NX+1)+i];
  double VX10 = VX[j*(NX+1)+i+1];
  double VX01 = VX[(j+1)*(NX+1)+i];
  double VX11 = VX[(j+1)*(NX+1)+i+1];
  vx = VX00*w00 + VX10*w10 + VX01*w01 + VX11*w11;

  double VY00 = VY[j*(NX+1)+i];
  double VY10 = VY[j*(NX+1)+i+1];
  double VY01 = VY[(j+1)*(NX+1)+i];
```

```
  double VY11 = VY[(j+1)*(NX+1)+i+1];
  vy = VY00*w00 + VY10*w10 + VY01*w01 + VY11*w11;

  return;
}


// ELLAM: assemble the global coefficient matrix
// GlbMatNdx: the auxiliary index matrix for the 9-banded coeff. matrix

void ELLAM::asmGlbMat(double *GlbMat, int *GlbMatNdx)
{
  // FILE *fp;
  int i, j, ie, je, k, l, nbr[4][4];
  int lblVrtx[4];
  double val, xel, yel;
  double RMM[4][4], RSMX[4][4], RSMY[4][4];
  double EMM[4][4], ESM[4][4], eltMat[4][4];

  // Reference Q1 element mass matrix (RMM)

  RMM[0][0] = 4;  RMM[0][1] = 2;  RMM[0][2] = 2;  RMM[0][3] = 1;
  RMM[1][0] = 2;  RMM[1][1] = 4;  RMM[1][2] = 1;  RMM[1][3] = 2;
  RMM[2][0] = 2;  RMM[2][1] = 1;  RMM[2][2] = 4;  RMM[2][3] = 2;
  RMM[3][0] = 1;  RMM[3][1] = 2;  RMM[3][2] = 2;  RMM[3][3] = 4;

  for (j=0; j<4; ++j)
    for (i=0; i<4; ++i)
      RMM[i][j] *= 1.0/36;

  // Reference Q1 element stiffness matrix -- X part (RSMX)

  RSMX[0][0] =  2;  RSMX[0][1] =  1;  RSMX[0][2] = -2;  RSMX[0][3] = -1;
  RSMX[1][0] =  1;  RSMX[1][1] =  2;  RSMX[1][2] = -1;  RSMX[1][3] = -2;
  RSMX[2][0] = -2;  RSMX[2][1] = -1;  RSMX[2][2] =  2;  RSMX[2][3] =  1;
  RSMX[3][0] = -1;  RSMX[3][1] = -2;  RSMX[3][2] =  1;  RSMX[3][3] =  2;

  for (j=0; j<4; ++j)
```

```
  for (i=0; i<4; ++i)
    RSMX[i][j] *= 1.0/6;

// Reference Q1 element stiffness matrix -- Y part (RSMY)

RSMY[0][0] =  2;  RSMY[0][1] = -2;  RSMY[0][2] =  1;  RSMY[0][3] = -1;
RSMY[1][0] = -2;  RSMY[1][1] =  2;  RSMY[1][2] = -1;  RSMY[1][3] =  1;
RSMY[2][0] =  1;  RSMY[2][1] = -1;  RSMY[2][2] =  2;  RSMY[2][3] = -2;
RSMY[3][0] = -1;  RSMY[3][1] =  1;  RSMY[3][2] = -2;  RSMY[3][3] =  2;

for (j=0; j<4; ++j)
  for (i=0; i<4; ++i)
    RSMY[i][j] *= 1.0/6;

// Neighbors' column indices in the 9-band global matrix

nbr[0][0] = 4;  nbr[0][1] = 5;  nbr[0][2] = 7;  nbr[0][3] = 8;
nbr[1][0] = 3;  nbr[1][1] = 4;  nbr[1][2] = 6;  nbr[1][3] = 7;
nbr[2][0] = 1;  nbr[2][1] = 2;  nbr[2][2] = 4;  nbr[2][3] = 5;
nbr[3][0] = 0;  nbr[3][1] = 1;  nbr[3][2] = 3;  nbr[3][3] = 4;

// Setzero

for (k=0; k<(NX+1)*(NY+1); ++k) {
  for (l=0; l<9; ++l) {
    GlbMat[k*9+l] = 0.0;
    GlbMatNdx[k*9+l] = -1;
  }
}

// Assemble the global coefficient matrix

for (je=1; je<=NY; ++je) {
  yel = Y[je] - Y[je-1];
  for (ie=1; ie<=NX; ++ie) {
    xel = X[ie] - X[ie-1];
    for (j=0; j<4; ++j) {
      for (i=0; i<4; ++i) {
```

```
          EMM[i][j] = (xel*yel)*RMM[i][j];
          ESM[i][j] = (yel/xel)*RSMX[i][j] + (xel/yel)*RSMY[i][j];
          eltMat[i][j] = EMM[i][j] + (Deltat*diff)*ESM[i][j];
        }
      }
      lblVrtx[0] = (je-1)*(NX+1) + (ie-1);
      lblVrtx[1] = lblVrtx[0] + 1;
      lblVrtx[2] = lblVrtx[0] + (NX+1);
      lblVrtx[3] = lblVrtx[2] + 1;
      for (j=0; j<4; ++j) {
        for (i=0; i<4; ++i) {
          val = GlbMat[lblVrtx[i]*9+nbr[i][j]];
          GlbMat[lblVrtx[i]*9+nbr[i][j]] = val + eltMat[i][j];
          GlbMatNdx[lblVrtx[i]*9+nbr[i][j]] = lblVrtx[j];
        }
      }
    }
  }

/*
  fp = fopen("GlbMat.dat", "w");
  for (i=0; i<(NX+1)*(NY+1); ++i) {
    fprintf(fp, "%6d  ", i);
    for (j=0; j<9; ++j)  fprintf(fp, "%14.6f  ", GlbMat[i*9+j]);
    fprintf(fp, "\n");
  }
  fclose(fp);
*/

  return;
}


// ELLAM: tracking characteristics based on the explicit Euler
// 40 micro steps

void ELLAM::trknExEuler(double &xstar, double &ystar,
  double xn, double yn, double tn, double tn1)
```

```
{
  int m;
  double deltat = (tn1-tn)/40;
  double vx, vy, xm, xm1, ym, ym1;

  xm = xn;
  ym = yn;

  for (m=1; m<=40; ++m) {
    velxy(vx, vy, xm, ym, tn);
    xm1 = xm + vx*deltat;
    ym1 = ym + vy*deltat;
    xm = xm1;
    ym = ym1;
  }

  xstar = xm1;
  ystar = ym1;

  return;
}


// ELLAM: assembling the global right-hand side for the old mass
// Un1: numerical solution at time t_{n-1}

void ELLAM::asmGlbRHS4OldMass(double *GlbRHS, double *Un1, double tn)
{
  // FILE *fp;
  int i, j, k, ie, je, kq, istar, jstar;
  int lblVrtx[4];
  double x0, y0, x1, y1, xc, yc, xel, yel;
  double intgrl, xstar, ystar, xx0, xx1, yy0, yy1;
  double U00, U10, U01, U11, vx, vy, w00, w10, w01, w11;
  double xq[9], yq[9], Ustar[9], phiq[2][2][9], eltRHS[4];

  for (je=1; je<=NY; ++je) {
    y0 = Y[je-1];  y1 = Y[je];  yc = 0.5*(y0+y1);  yel = y1 - y0;
```

```
for (ie=1; ie<=NX; ++ie) {
  x0 = X[ie-1];  x1 = X[ie];  xc = 0.5*(x0+x1);  xel = x1 - x0;

  // Global indices of the four vertices
  lblVrtx[0] = (je-1)*(NX+1) + (ie-1);
  lblVrtx[1] = lblVrtx[0] + 1;
  lblVrtx[2] = lblVrtx[0] + (NX+1);
  lblVrtx[3] = lblVrtx[2] + 1;

  // Gaussian quadrature points and test function values
  for (kq=0; kq<GQR3npts; ++kq) {
    xq[kq] = xc + (xel/2)*GQR3wab[kq][1];
    yq[kq] = yc + (yel/2)*GQR3wab[kq][2];
    phiq[0][0][kq] = ((x1-xq[kq])/(x1-x0))*((y1-yq[kq])/(y1-y0));
    phiq[1][0][kq] = ((xq[kq]-x0)/(x1-x0))*((y1-yq[kq])/(y1-y0));
    phiq[0][1][kq] = ((x1-xq[kq])/(x1-x0))*((yq[kq]-y0)/(y1-y0));
    phiq[1][1][kq] = ((xq[kq]-x0)/(x1-x0))*((yq[kq]-y0)/(y1-y0));
  }

  // Back-tracking the quadrature points
  for (kq=0; kq<GQR3npts; ++kq) {
    velxy(vx, vy, xq[kq], yq[kq], tn);
    // vx = velx(xq[kq], yq[kq], tn);
    // vy = vely(xq[kq], yq[kq], tn);
    xstar = xq[kq] - vx*Deltat;
    ystar = yq[kq] - vy*Deltat;


    if (xstar<=XA || xstar>=XB || ystar<=YC || ystar>=YD) {
      Ustar[kq] = 0;
      continue;
    }

    istar = (int)floor((xstar-XA)/Deltax);
    jstar = (int)floor((ystar-YC)/Deltay);
    xx0 = X[istar];  xx1 = X[istar+1];
    yy0 = Y[jstar];  yy1 = Y[jstar+1];
```

```
        U00 = Un1[jstar*(NX+1)+istar];
        U10 = Un1[jstar*(NX+1)+istar+1];
        U01 = Un1[(jstar+1)*(NX+1)+istar];
        U11 = Un1[(jstar+1)*(NX+1)+istar+1];
        w00 = ((xx1-xstar)/(xx1-xx0))*((yy1-ystar)/(yy1-yy0));
        w10 = ((xstar-xx0)/(xx1-xx0))*((yy1-ystar)/(yy1-yy0));
        w01 = ((xx1-xstar)/(xx1-xx0))*((ystar-yy0)/(yy1-yy0));
        w11 = ((xstar-xx0)/(xx1-xx0))*((ystar-yy0)/(yy1-yy0));
        Ustar[kq] = U00*w00 + U10*w10 + U01*w01 + U11*w11;
      }

      // Old mass
      k = 0;
      for (j=0; j<2; ++j) {
        for (i=0; i<2; ++i) {
          intgrl = 0.0;
          for (kq=0; kq<GQR3npts; ++kq)
            intgrl += Ustar[kq]*phiq[i][j][kq]*GQR3wab[kq][0];
          intgrl *= xel*yel;
          eltRHS[k] = intgrl;
          k++;
        }
      }

      // Assembling the element RHS into the global RHS
      for (k=0; k<4; ++k)  GlbRHS[lblVrtx[k]] += eltRHS[k];
    }
  }

  return;
}


// ELLAM: assembling the global right-hand side for the source term

void ELLAM::asmGlbRHS4Srs(double *GlbRHS, double tn)
{
  int i, j, k, ie, je, kq;
```

```
int lblVrtx[4];
double intgrl, x0, y0, x1, y1, xc, yc, xel, yel;
double xq[9], yq[9], fq[9], phiq[2][2][9], eltRHS[4];

for (je=1; je<=NY; ++je) {
  y0 = Y[je-1];  y1 = Y[je];  yc = 0.5*(y0+y1);  yel = y1 - y0;
  for (ie=1; ie<=NX; ++ie) {
    x0 = X[ie-1];  x1 = X[ie];  xc = 0.5*(x0+x1);  xel = x1 - x0;

    // Global indices of the four vertices
    lblVrtx[0] = (je-1)*(NX+1) + (ie-1);
    lblVrtx[1] = lblVrtx[0] + 1;
    lblVrtx[2] = lblVrtx[0] + (NX+1);
    lblVrtx[3] = lblVrtx[2] + 1;

    // Gaussian quadrature points and function values
    for (kq=0; kq<GQR3npts; ++kq) {
      xq[kq] = xc + (xel/2)*GQR3wab[kq][1];
      yq[kq] = yc + (yel/2)*GQR3wab[kq][2];
      phiq[0][0][kq] = ((x1-xq[kq])/(x1-x0))*((y1-yq[kq])/(y1-y0));
      phiq[1][0][kq] = ((xq[kq]-x0)/(x1-x0))*((y1-yq[kq])/(y1-y0));
      phiq[0][1][kq] = ((x1-xq[kq])/(x1-x0))*((yq[kq]-y0)/(y1-y0));
      phiq[1][1][kq] = ((xq[kq]-x0)/(x1-x0))*((yq[kq]-y0)/(y1-y0));
      fq[kq] = srs(xq[kq], yq[kq], tn);
    }

    // Element RHS from the source term
    k = 0;
    for (j=0; j<2; ++j) {
      for (i=0; i<2; ++i) {
        intgrl = 0.0;
        for (kq=0; kq<GQR3npts; ++kq)
          intgrl += fq[kq]*phiq[i][j][kq]*GQR3wab[kq][0];
        intgrl *= xel*yel;
        eltRHS[k] = Deltat*intgrl;
        k++;
      }
    }
```

```
      // Assembling the element RHS into the global RHS
      for (k=0; k<4; ++k)  GlbRHS[lblVrtx[k]] += eltRHS[k];
    }
  }

  return;
}


// ELLAM: modify the global system based on
// a homogeneous Dirichlet boundary condition

void ELLAM::modGlbSysDirichletBndryCond(double *GlbMat, double *GlbRHS)
{
  int i, j, k, l;

  // cout << "In modGlbSysDirichletBndryCond...\n";

  j = 0;  // Domain bottom side
  for (i=0; i<=NX; ++i) {
    k = j*(NX+1) + i;
    for (l=0; l<9; ++l)  GlbMat[k*9+l] = 0.0;
    GlbMat[k*9+4] = 1.0;
    GlbRHS[k] = 0.0;
  }

  j = NY;  // Domain top side
  for (i=0; i<=NX; ++i) {
    k = j*(NX+1) + i;
    for (l=0; l<9; ++l)  GlbMat[k*9+l] = 0.0;
    GlbMat[k*9+4] = 1.0;
    GlbRHS[k] = 0.0;
  }

  i = 0;  // Domain left side
  for (j=1; j<=(NY-1); ++j) {
    k = j*(NX+1) + i;
```

```
    for (l=0; l<9; ++l)  GlbMat[k*9+l] = 0.0;
    GlbMat[k*9+4] = 1.0;
    GlbRHS[k] = 0.0;
  }

  i = NX;  // Domain right side
  for (j=1; j<=(NY-1); ++j) {
    k = j*(NX+1) + i;
    for (l=0; l<9; ++l)  GlbMat[k*9+l] = 0.0;
    GlbMat[k*9+4] = 1.0;
    GlbRHS[k] = 0.0;
  }

  return;
}


// ELLAM: solve the nonsymmetric global linear system
// by BiCGStab with the simple diagonal preconditioner
// a specialized version
// cf: p.27 of the SIAM Templates book

int ELLAM::slvGlbSysBiCGStab(int n, double *x,
  double *A, int *M, double *b, double tol)
{
  int i, j, k, kmax;
  double alpha, beta, omega, res, rho1, rho2, dpst, dptt;
  double *B, *p, *phat, *r, *rhat, *s, *shat, *t, *v;

  B = new double[n];
  p = new double[n];
  r = new double[n];
  s = new double[n];
  t = new double[n];
  v = new double[n];
  phat = new double[n];
  rhat = new double[n];
  shat = new double[n];
```

```
kmax = n;

for (i=0; i<n; ++i)  B[i] = A[i*9+4];

for (i=0; i<n; ++i) {
  r[i] = b[i];
  for (j=0; j<9; ++j)  r[i] -= A[i*9+j]*x[M[i*9+j]];
}

for (i=0; i<n; ++i)  rhat[i] = r[i];

res = 0;
for (i=0; i<n; ++i)  res += r[i]*r[i];

// tol *= res;

for (k=1; k<=kmax; ++k) {
  rho1 = 0;
  for (i=0; i<n; ++i)  rho1 += r[i]*rhat[i];

  if (rho1==0)  {return 2;}

  if (k==1) {
    for (i=0; i<n; ++i)  p[i] = r[i];
  } else {
    beta = (rho1/rho2)*(alpha/omega);
    for (i=0; i<n; ++i) {
      p[i] -= omega*v[i];
      p[i] = r[i] + beta*p[i];
    }
  }

  for (i=0; i<n; ++i)  phat[i] = p[i]/B[i];

  for (i=0; i<n; ++i) {
    v[i] = 0;
    for (j=0; j<9; ++j)  v[i] += A[i*9+j]*phat[M[i*9+j]];
```

```
}

alpha = 0;
for (i=0; i<n; ++i)  alpha += rhat[i]*v[i];
alpha = rho1/alpha;

for (i=0; i<n; ++i)  s[i] = r[i] - alpha*v[i];

res = 0;
for (i=0; i<n; ++i)  res += v[i]*v[i];

if (res<tol) {
  for (i=0; i<n; ++i)  x[i] += alpha*phat[i];
  return 0;
}

for (i=0; i<n; ++i)  shat[i] = s[i]/B[i];

for (i=0; i<n; ++i) {
  t[i] = 0;
  for (j=0; j<9; ++j)  t[i] += A[i*9+j]*shat[M[i*9+j]];
}

dpst = 0;
dptt = 0;
for (i=0; i<n; ++i) {
  dpst += s[i]*t[i];
  dptt += t[i]*t[i];
}
omega = dpst/dptt;

for (i=0; i<n; ++i) {
  x[i] += alpha*phat[i] + omega*shat[i];
  r[i] = s[i] - omega*t[i];
}

rho2 = rho1;
```

```
    res = 0;
    for (i=0; i<n; ++i)  res += r[i]*r[i];

    if (res<tol)  {return 0;}
    if (omega==0)  {return 3;}
  }

  delete[] B, p, phat, r, rhat, s, shat, t, v;
  return 1;
}


// ELLAM: set an initial condition

void ELLAM::setInitCond(double *U, int nx, int ny, int order)
{
  int i, j, k;

  if (nx!=NX || ny!=NY) {
    cout << "Mesh mismatch!\n";
    exit(-1);
  }

  U0 = new double[(NX+1)*(NY+1)];

  if (order==0) {
    for (j=0; j<=NY; ++j) {
      for (i=0; i<=NX; ++i) {
        k = j*(NX+1) + i;
        U0[k] = U[k];
      }
    }
  }

  return;
}
```

```
// ELLAM: get the final solution

void ELLAM::getFinalSln(double *U, int nx, int ny, int order)
{
  int i, j, k;

  if (nx!=NX || ny!=NY) {
    cout << "Mesh mismatch!\n";
    exit(-1);
  }

  if (order==0) {
    for (j=0; j<=NY; ++j) {
      for (i=0; i<=NX; ++i) {
        k = j*(NX+1) + i;
        U[k] = UT[k];
      }
    }
  }

  return;
}


// ELLAM: execution (time-marching)
// verbosity:  0: none, 1: minimum, 2: medium

void ELLAM::execute(int verbosity)
{
  FILE *fp;
  int i, j, n, szGlbSys;
  double tn, tn1;
  double *GlbMat, *GlbRHS, *sln, *Un, *Un1, *Utmp;
  int *GlbMatNdx;

  szGlbSys = (NX+1)*(NY+1);

  sln = new double[szGlbSys];
```

```
Un = new double[szGlbSys];
Un1 = new double[szGlbSys];
GlbMat = new double[szGlbSys*9];
GlbMatNdx = new int[szGlbSys*9];
GlbRHS = new double[szGlbSys];

asmGlbMat(GlbMat, GlbMatNdx);

Un1 = U0;
if (verbosity>0)  cout << "ELLAM time-marching...\n";
for (n=1; n<=NT; ++n) {
  tn = n*Deltat;
  tn1 = (n-1)*Deltat;
  if (verbosity>0)  cout << "n=" << n << "  ";

  for (i=0; i<szGlbSys; ++i)  GlbRHS[i] = 0;
  asmGlbRHS4Srs(GlbRHS, tn);
  asmGlbRHS4OldMass(GlbRHS, Un1, tn);
  modGlbSysDirichletBndryCond(GlbMat, GlbRHS);

  for (i=0; i<szGlbSys; ++i)  sln[i] = Un1[i];
  slvGlbSysBiCGStab(szGlbSys, sln, GlbMat, GlbMatNdx, GlbRHS, 1E-30);
  for (i=0; i<szGlbSys; ++i)  Un[i] = sln[i];

  // Refreshing:  swap pointers!
  Utmp = Un1;  Un1 = Un;  Un = Utmp;
}
if (verbosity>0)  cout << "\n";
UT = Un1;

fp = fopen("GlbMat.dat", "w");
for (i=0; i<(NX+1)*(NY+1); ++i) {
  fprintf(fp, "%6d  ", i);
  for (j=0; j<9; ++j)  fprintf(fp, "%14.6f  ", GlbMat[i*9+j]);
  fprintf(fp, "\n");
}
fclose(fp);
```

```
  fp = fopen("GlbRHS.dat", "w");
  for (i=0; i<(NX+1)*(NY+1); ++i)
    fprintf(fp, "%6d  %14.6f\n", i, GlbRHS[i]);
  fclose(fp);

  delete[] GlbMat, GlbMatNdx, GlbRHS, sln, Un, Un1;
  return;
}
```

4.3 Mini ELLAM: Examples

The C++ code below has two examples (worked in the previous Chapter).

```
// TstMiniELLAM.cpp
// A small test program for MiniELLAM
// Jiangguo (James) Liu,  ColoState,  12/2008 - 03/2009

#include <cmath>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include "MiniELLAM.h"
using namespace std;


double ex0sln(double x, double y, double t)
{
  double a = 1;  double b = 1;
  double xc = -0.5;  double yc = -0.5;
  double sigma122 = 50;
  double xt = x - xc - a*t;
  double yt = y - yc - b*t;
  double u = exp(-(xt*xt+yt*yt)*sigma122);
  return u;
}
```

```
double ex0velx(double x, double y, double t)
{
  double a = 1.0;  return a;
}

double ex0vely(double x, double y, double t)
{
  double b = 1.0;  return b;
}

double ex0srs(double x, double y, double t)
{
  double D = 1E-3;
  double a = 1;  double b = 1;
  double xc = -0.5;  double yc = -0.5;
  double sigma212 = 200;  double sigma14 = 10000;
  double xt = x - xc - a*t;
  double yt = y - yc - b*t;
  double u = ex0sln(x, y, t);
  double f = D*(sigma212-sigma14*(xt*xt+yt*yt))*u;
  return f;
}


double ex1sln(double x, double y, double t)
{
  double sigma22 = 4E-3;  double D = 1E-4;
  double xc = -0.5;  double yc = 0.0;
  double xstar = cos(4*t)*x + sin(4*t)*y;
  double ystar = -sin(4*t)*x + cos(4*t)*y;
  double xt = xstar - xc;
  double yt = ystar - yc;
  double u = exp(-(xt*xt+yt*yt)/(sigma22+4*D*t));
  u = sigma22/(sigma22+4*D*t)*u;
  return u;
}

double ex1velx(double x, double y, double t)  {return -4*y;}
```

```
double ex1vely(double x, double y, double t)  {return 4*x;}

double ex1srs(double x, double y, double t)  {return 0.0;}


void main()
{
  FILE *fp;
  int i, j, nt, nx, ny;
  double a, b, c, d, hx, hy, t0, tf;
  double *x, *y, *U;
  double *Ex1Vx, *Ex1Vy;
  double PI = 3.141592653589793;

  cout << "OK beginning!\n";

  a = -1;  b = 1;  c = -1;  d = 1;  nx = 128;  ny = 128;
  t0 = 0;  tf = PI/2;  nt = 50;

  hx = (b-a)/nx;
  x = new double[nx+1];
  x[0] = a;
  for (i=1; i<nx; ++i)  x[i] = x[i-1] + hx;
  x[nx] = b;

  hy = (d-c)/ny;
  y = new double[ny+1];
  y[0] = c;
  for (j=1; j<ny; ++j)  y[j] = y[j-1] + hy;
  y[ny] = d;

  Ex1Vx = new double[(nx+1)*(ny+1)];
  Ex1Vy = new double[(nx+1)*(ny+1)];
  for (j=0; j<=ny; ++j) {
    for (i=0; i<=nx; ++i) {
      Ex1Vx[j*(nx+1)+i] = ex1velx(x[i], y[j], t0);
      Ex1Vy[j*(nx+1)+i] = ex1vely(x[i], y[j], t0);
```

```
    }
  }

  U = new double[(nx+1)*(ny+1)];
  for (j=0; j<=ny; ++j)
    for (i=0; i<=nx; ++i)
      U[j*(nx+1)+i] = ex1sln(x[i], y[j], t0);

  fp = fopen("U0.dat", "w");
  for (j=0; j<=ny; ++j)
    for (i=0; i<=nx; ++i)
      fprintf(fp, "%12.6f  %12.6f  %12.6f\n", x[i], y[j], U[j*(nx+1)+i]);
  fclose(fp);

  ELLAM ellam;
  ellam.setMesh(a, b, c, d, nx, ny);
  ellam.setTime(t0, tf, nt);
  // ellam.setVelFxns(ex1velx, ex1vely);
  ellam.setVelData(Ex1Vx, Ex1Vy, nx, ny, 0);
  ellam.setDiffConst(1E-4);
  ellam.setSrsFxn(ex1srs);
  ellam.setInitCond(U, nx, ny, 0);
  ellam.execute(1);
  ellam.getFinalSln(U, nx, ny, 0);

  fp = fopen("UT.dat", "w");
  fprintf(fp, "%5d   %5d\n", nx, ny);
  for (j=0; j<=ny; ++j)
    for (i=0; i<=nx; ++i)
      fprintf(fp, "%12.6f  %12.6f  %12.6f\n", x[i], y[j], U[j*(nx+1)+i]);
  fclose(fp);

  delete[] Ex1Vx, Ex1Vy;
  delete[] U, x, y;
  cout << "Happy ending!\n";
  return;
}
```

```
% PlotUT.m
% Plotting 3-dim graph for numerical solution UT
% Jiangguo (James) Liu,  ColoState, 02/2009

clc;
clear all;
close all;

fp = fopen('UT.dat', 'r');

v = zeros(2,1);
v = fscanf(fp, '%d', 2);

NX = v(1);
NY = v(2);

X = zeros(NX+1,1);
Y = zeros(NY+1,1);
UT = zeros(NX+1, NY+1);

v = zeros(3,1);

for j = 0 : NY
  for i = 0 : NX
    v = fscanf(fp, '%f', 3);
    X(i+1) = v(1);
    Y(j+1) = v(2);
    UT(i+1,j+1) = v(3);
    end
end

fclose(fp);

figure(1);
mesh(X, Y, UT);
%  title('Approx Solution UT');
%  zlim([-0.1, 1.1]);
```

```
print -f1 UT.ps

disp('pause...');
pause;
close all;
```

Appendix A

List of Abbreviations

The following list might be helpful for understanding my code or other documents.

abs absolute / absolute value

aprx approximate / approximation

asm assemble

avg average

bd bound

bdd bounded

bdry boundary

bk back / backward

bkp backup

Cart Cartesian

chara characteristic(s)

cmplt complete

cntr center

comput compute

cond condition

const constant

crd coordinate

def definition

deriv derivative

diam diameter

diff differentiation / diffusion

dist distance

dom domain

elt element

eqn equation

err error

fwd forward

fxn function

fxnl functional

gen general / generate

geom geometry

glb global

grp group

idx index (see also **ind**,**ndx**)

ind index (see also **idx**,**ndx**)

init initialization

int integer

integ integral / integrand / integration

intvl interval

IO inout / outpout

itr iteration

lbl label

lcl local

lin linear

loc locate / location

lvl level

mat matrix

max maximal / maximum

mdl middle

min minimal / minimum

nbr neighbor

nd node

ndx index (see also **ind**, **idx**)

nlin nonlinear

num number / numerical

pt point

ptr pointer

quad quadrature

rect rectangle / rectangular

ref reference

rhs,**RHS** right-hand side

seg segment(s)

sln solution

slv solve

sqr square

sqrt square root

srs source

stf stiffness

sub sub-

sys system

tol tolerance

trl trial

trk tracking

tst test

tyme time (to avoid conflicts with "time", which might be reserved in some languages)

val value

vec vector

vel velocity

vrtx vertex / vortex

whl whole

wt weight

Bibliography

[1] M.A. Celia, T.F. Russell, I. Herrera, and R.E. Ewing, *An Eulerian-Lagrangian localized adjoint method for the advection-diffusion equation*, Adv. Water Resour., **13** (1990), 187-206.

[2] G.H. Golub and C.F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, 3rd ed., 1996.

[3] J. Liu, *Efficient numerical techniques for advection dominated transport equations*, Ph.D. Thesis, University of South Carolina, 2001.

[4] J. Liu, B. Popov, H. Wang, and R.E. Ewing, *Convergence analysis of wavelet schemes for convection-reaction equations under minimal regularity assumptions*, SIAM J. Numer. Anal., *To appear.*

[5] W. Schroeder, K. Martin, and B. Lorensen, *The visualization toolkit: An objected-oriented approach to 3D graphics*, 3rd ed., Kitware, 2002.

[6] H. Wang, H.K. Dahle, R.E. Ewing, M.S. Espedal, R.C. Sharpley, and S. Man, *An ELLAM scheme for advection-diffusion equations in two dimensions*, SIAM J. Sci. Comput., **20** (1999), 2160-2194.