Implementation and performance of a three-dimensional numerical transport model

B.P. Sommeijer, J. Kok

Department of Numerical Mathematics

Report NM-R9402 February 1994

Implementation and Performance of a Three-Dimensional Numerical Transport Model

B.P. Sommeijer and J. Kok
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract
We present two numerical solution methods for the time integration of a three-dimensional model for computing transport of salinity, pollutants, suspended material (such as sediment or mud), etc. in shallow seas. The emphasis in this paper is on the performance of the methods when implemented on a vector/parallel, shared memory computer, such as a CRAY type machine. The first method is an explicit time integrator and can straightforwardly be vectorized and parallelized. Although a stabilizing technique has been applied to this method, it still suffers from a severe timestep restriction. The second method is partly implicit, resulting in much better stability characteristics; however, as a consequence of the implicitness, it requires in each step the solution of a large number of tridiagonal systems. When implemented in a standard way, the recursive nature would prevent vectorization, resulting in a very long solution time. Following a suggestion of Golub and Van Loan, this part of the algorithm has been tuned for use on the CRAY Y-MP4/464. On the basis of a large-scale test problem, performance results will be presented for various implementations.

AMS Subject Classification (1991): 65M12, 65M20
CR Subject Classification (1991): G.1.7, G.1.8
Keywords and Phrases: Transport models, 3D advection-diffusion equations, numerical time integration, vectorization, parallel processing.
Note: This work was sponsored by the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF) who provided the authors with a grant from Cray Research University Grants Program.

1. Introduction

In [8], several time integration techniques have been studied for the efficient solution of the 3D advection-diffusion-reaction equations modelling the transport of salinity, pollutants, suspended material (such as sediment or mud), etc. in shallow seas. The conclusion formulated in [8] was that stabilized, explicit Runge-Kutta methods may be eligible candidates because they are well suited to efficiently exploit the vectorization and parallelization facilities offered by CRAY type computers. However, as a disadvantage, these methods have to obey a rather restrictive condition on the timestep because of stability reasons. A remedy to avoid the stability condition is offered by the Odd-Even Line Hopscotch method. Since this method is partly implicit, a large number of uncoupled tridiagonal systems has to be solved in each step. Normally, the straightforward and numerically stable way to solve these systems does not allow to exploit vectorization facilities, due to the recursive nature of the standard decomposition approach. Since the systems are uncoupled and of identical shape (which includes a similar storing of the matrix elements and the right-hand side components), it is possible to vectorize (over all the tridiagonal systems) each step within the standard decomposition algorithm. This technique, which will be termed vectorization-across-the-tridiagonal-systems, was originally proposed by Golub and Van Loan [4] and has also been used by De Goede [3] in his 3D hydrodynamical shallow water model. This vectorization technique results in a considerably increased performance. Moreover, it can straightforwardly be combined with parallelization for use on a multi (vector)processor system; the *atexpert* utility [2] (to predict performance on a dedicated system) indicates almost optimal speed-up (more than 90% efficiency). When equipped with this linear system solver, the hopscotch method turns out to be superior to the stabilized Runge-Kutta methods.

The paper is organized as follows: in Section 2, we will discuss the actual transport model that has been studied. Section 3 is devoted to the discretization of this PDE. First, we briefly outline the spatial (or, semi-) discretization method and the consequences for the data structures that we need, as well as the resulting vectorization capabilities. Next, two time integration methods are described in some detail, including the motivations for their choice. In Section 4, we survey the various possibilities to solve the linear systems occurring in the Hopscotch integrator and discuss several options to efficiently cope with the Jacobian matrices. Furthermore, the actual implementation of the solver is discussed. Its performance is illustrated in Section 5 by applying it to a large scale test example. Conclusions are formulated in Section 6.

Report NM-R9402
ISSN 0169-0388
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

2. Definition of the mathematical model

The mathematical model describing transport processes in three dimensions is given by the system of advection-diffusion-reaction equations [9]

$$(2.1)\qquad \frac{\partial c_i}{\partial t} = -\frac{\partial}{\partial x}(uc_i) - \frac{\partial}{\partial y}(vc_i) - \frac{\partial}{\partial z}((w - w_f)c_i) + \frac{\partial}{\partial x}\Big(\varepsilon_x \frac{\partial c_i}{\partial x}\Big) + \frac{\partial}{\partial y}\Big(\varepsilon_y \frac{\partial c_i}{\partial y}\Big) + \frac{\partial}{\partial z}\Big(\varepsilon_z \frac{\partial c_i}{\partial z}\Big) + g_i,$$

where c_i are the unknown concentrations of the contaminants. The local fluid velocities u, v, w (in x, y, z directions, respectively) have to be provided by a hydrodynamical model; since this paper merely discusses the solution of (2.1), the velocity field is considered to be known in advance. The fall velocity w_f, which may be a nonlinear function of the concentration, is only relevant in the case of modelling transport of suspended material. The terms g_i describe chemical reactions, emissions from sources, etc. and therefore depend on the concentrations c_i. Thus the mutual coupling of the equations in the system (2.1) is only due to the functions g_i. Because of this weak coupling, we shall confine ourselves to the numerical modelling of a single transport equation (and omit the subscript i in the sequel), since the extension to systems is relatively straightforward. Finally, the diffusion coefficients ε_x, ε_y, and ε_z are assumed to be given functions.

The physical domain in space is bounded by vertical, closed boundary planes, by the water elevation surface, and by the bottom profile. On these boundaries, Dirichlet, Neumann or mixed boundary conditions will be prescribed. Supplementing this with an initial condition, the concentration c can be computed in space and time.

3. Discretization methods

To arrive at a fully discrete numerical approximation, we will follow the method of lines (MOL) approach. That is, we first transform equation (2.1) into a system of ordinary differential equations (ODEs) by discretizing only the spatial derivatives and leaving time continuous. Next, these ODEs will be integrated in time numerically. In the following two subsections, both steps in this discretization process will be discussed separately.

3.1. Spatial discretization

Typically, the physical domain in space is quite irregular, both in horizontal and vertical direction. One possible way to cope with this situation is to map the physical domain on a rectangular box by means of coordinate transformations. The advantage of this approach is that the physical boundaries can be exactly represented; however, a serious disadvantage is a much more complicated mathematical model and, additionally, the danger of introducing coefficients of large magnitude. This increases the stiffness in the resulting ODEs and excludes the possibility to use explicit time integration techniques (see [8] for a discussion on this aspect).

Therefore, we decided to follow the so-called 'dummy-point' approach. By this we mean that the whole physical domain will be enclosed by a rectangular box. Obviously, the disadvantage is that we may introduce many artificial (meaningless) points. However, the regular grid structure allows for an efficient implementation on vector computers, which compensates for the additional computational effort.

Before applying the semi-discretization process, it is convenient to rewrite equation (2.1) by taking into account the particular applications we have in mind. In this paper, we shall only consider the *incompressible* case, that is, we assume $u_x + v_y + w_z = 0$; furthermore, the diffusion coefficients ε_x, ε_y, ε_z are assumed to be constant and equal to ε. Recalling that we concentrate on a single transport equation, (2.1) simplifies to

$$(3.1)\qquad \frac{\partial c}{\partial t} = -u\frac{\partial}{\partial x}c - v\frac{\partial}{\partial y}c - w\frac{\partial}{\partial z}c + \frac{\partial}{\partial z}(w_f c) + \varepsilon\Big(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2} + \frac{\partial^2 c}{\partial z^2}\Big) + g.$$

The first step in the semi-discretization process is to let the physical domain be enclosed by a rectangular box, containing the grid points

$$(3.2a)\qquad P_{j,k,m} := (x_0 + j\Delta x,\, y_0 + k\Delta y,\, z_0 + m\Delta z),\quad j = 0, \dots, J+1;\quad k = 0, \dots, K+1;\quad m = 0, \dots, M+1,$$

where (x_0, y_0, z_0) corresponds with the south-west corner point at the bottom of the rectangular box. It will be assumed that the mesh parameters Δx, Δy and Δz are such that the physical boundaries can be approximated by a subset of grid points with sufficient accuracy. This subset of boundary grid points will be denoted by $\partial\mathbb{B}$ and divides the remaining grid points in outer and inner ones lying 'outside' and 'inside' $\partial\mathbb{B}$. These sets of points are denoted by $\mathbb{B}_{out}$ and $\mathbb{B}_{in}$, respectively. Furthermore, we assume that the enclosing box is such that no grid points of $\partial\mathbb{B}$ are on the boundary planes of the box (i.e., the indices j, k, m of the boundary points satisfy $1 \le j \le J$, $1 \le k \le K$, and $1 \le m \le M$).

Next the spatial differential operators $\partial/\partial x$, $\partial/\partial y$ and $\partial/\partial z$ are replaced by (standard) *symmetric* differences. We remark that there is a considerable amount of literature on the spatial discretization of advection dominated partial differential equations. For a recent overview we refer to [10], in which also *unsymmetric* discretizations (such as upwind) are discussed. Furthermore, we denote the numerical approximations at $P_{j,k,m}$ to c, u, ... by capitals $C_{j,k,m}$, $U_{j,k,m}$, ... , and we introduce the spatial shift operators $\mathrm{X}_\pm$, $\mathrm{Y}_\pm$ and $\mathrm{Z}_\pm$ defined by

$$\mathrm{X}_{\pm}C_{j,k,m} := C_{j\pm1,k,m}\,, \quad \mathrm{Y}_{\pm}C_{j,k,m} := C_{j,k\pm1,m}\,, \quad \mathrm{Z}_{\pm}C_{j,k,m} := C_{j,k,m\pm1}\,.$$

Then, on the *computational* set of grid points $\mathbb{S}$ defined by

(3.2b) $$\mathbb{S} := \{P_{j,k,m}\colon\ 1 \le j \le J,\ \ 1 \le k \le K\,,\ \ 1 \le m \le M\}$$

the associated ODEs take the form

(3.3a) $$\frac{\mathrm{d}C_{j,k,m}}{\mathrm{d}t} = -\Big\{\frac{1}{2\Delta x}U_{j,k,m}[\mathrm{X}_+ - \mathrm{X}_-] + \frac{1}{2\Delta y}V_{j,k,m}[\mathrm{Y}_+ - \mathrm{Y}_-] + \frac{1}{2\Delta z}W_{j,k,m}[\mathrm{Z}_+ - \mathrm{Z}_-]\Big\}\,C_{j,k,m}$$
$$+\,\varepsilon\Big\{\frac{1}{(\Delta x)^2}\,[\mathrm{X}_+ - 2 + \mathrm{X}_-] + \frac{1}{(\Delta y)^2}\,[\mathrm{Y}_+ - 2 + \mathrm{Y}_-] + \frac{1}{(\Delta z)^2}\,[\mathrm{Z}_+ - 2 + \mathrm{Z}_-]\Big\}\,C_{j,k,m}$$
$$+\,\frac{1}{2\Delta z}\,\omega(C)_{j,k,m}\,[\mathrm{Z}_+ - \mathrm{Z}_-]\;C_{j,k,m} + g_{j,k,m},$$

where $\omega(c) := w_f(c) + \dfrac{c\,\partial w_f(c)}{\partial c}$.

The next step is to take into account the boundary conditions. If $P_{j,k,m}$ is a (physical) boundary point where the boundary condition is of Dirichlet-type, then $C_{j,k,m}$ is explicitly given, so that by means of differentiation with respect to time, we obtain ODEs of the form

(3.3b) $$\frac{\mathrm{d}C_{j,k,m}}{\mathrm{d}t} = d_{j,k,m}(t),$$

which should replace the ODEs occurring in (3.3a) at all Dirichlet-type boundary points.

If $P_{j,k,m}$ is a boundary point of non-Dirichlet-type, then the corresponding ODE in (3.3a) asks for the concentration at one or more outer grid points. By means of the boundary conditions, these auxiliary concentrations can explicitly be expressed in terms of concentrations at inner (or boundary) grid points. Finally, we assume 'dummy' concentrations at all points in the computational set $\mathbb{S}$ which are in $\mathbb{B}_{out}$.

In conclusion, the equations (3.3) corresponding to the set of grid points $\mathbb{S}$ as defined by (3.2b) define a second-order consistent semi-discretization of the initial-boundary value problem (2.1) of dimension $N := J{\cdot}K{\cdot}M$. Notice that only the concentrations defined by this system of ODEs corresponding to the grid points of $\mathbb{B}_{in}$ and $\partial\mathbb{B}$ are relevant.

In the analysis of time integrators for (3.3), it is more convenient to represent this system in the form

(3.4) $$\frac{\mathrm{d}C(t)}{\mathrm{d}t} = F(t,C(t)) := A_x(t)C(t) + A_y(t)C(t) + A_z(t)C(t) + W_z(C(t))C(t) + G(t,C(t)) + B_x(t) + B_y(t) + B_z(t),$$

where $C(t)$ is a vector of dimension N containing all concentrations defined at the points of $\mathbb{S}$, $A_x(t)$, $A_y(t)$ and $A_z(t)$ are N-by-N tridiagonal matrices only depending on t, and $W_z(C)$ is an N-by-N tridiagonal matrix which vanishes if the fall velocity w_f occurring in (2.1) vanishes. The N-dimensional vector $G(t,C(t))$ is the discrete analogue of the source function g, and the vectors $B_x(t)$, $B_y(t)$ and $B_z(t)$ represent the inhomogeneous contributions of the boundary conditions at the vertical east and west boundaries, at the vertical north and south boundaries, and at the surface and bottom boundaries, respectively.

Owing to the above 'dummy-point' approach, the datastructures in the code are extremely simple: the arrays U, V, and W (containing the velocity field), and the array DCDT (containing the time derivatives of the concentrations) are defined on the computational domain $\mathbb{S}$; hence, these three-times subscripted arrays have indices running from 1 to J, 1 to K, and 1 to M, respectively. The array C (containing the concentration vector C) is defined on the whole enclosing box, where the indices respectively run from 0 to J+1, 0 to K+1, and 0 to M+1. A consequence of this approach is that the whole array C will contain values, including the meaningless values corresponding to grid points lying in $\mathbb{B}_{out}$. Hence, on entering the subroutine F to calculate the time derivatives, the following steps have to be performed: (i) in the case of non-Dirichlet boundary conditions, the points in $\mathbb{B}_{out}$ which are on a distance of one grid point from $\partial\mathbb{B}$ should be calculated using the boundary conditions (notice that these points may lie on the boundary of the enclosing box); (ii) calculate the time derivatives of the concentration (i.e., evaluate the right-hand side of (3.3a)) in *all* points belonging to $\mathbb{S}$ and store these values into the array DCDT; (iii) in the case of Dirichlet boundary points, the corresponding values in DCDT have to be overwritten by the expressions defined in (3.3b). In this way, an efficient implementation of the subroutine F on a vector machine can be obtained (see the results in Section 6).

3.2. Time integration methods

In [8] several time integration techniques have been discussed and compared on the basis of their suitability to solve transport equations. It turned out that the explicit, stabilized Runge-Kutta methods and the Odd-Even Line Hopscotch method are the most promising for integrating the space-discretized transport equation (3.4) on CRAY-type computers. In the following subsections, these methods will be discussed in detail.

3.2.1. Stabilized Runge-Kutta methods

The q-stage Runge-Kutta method that we consider to advance the solution of (3.4) over one step of size Δt is defined by

$$\begin{aligned} &C^{(0)} = C_n, \\ (3.5)\qquad &C^{(j)} = C_n + \alpha_j \,\Delta t\, F(t_n + \mu_j\, \Delta t,\, C^{(j-1)}), \qquad j = 1, \ldots, q, \\ &C_{n+1} = C^{(q)}. \end{aligned}$$

In this method, the free parameters α_j and μ_j will be chosen to give the method the required properties: second-order accuracy, as large a stability boundary as possible, and minimal costs per step. The first requirement is fulfilled by setting $\alpha_q = 1$, $\alpha_{q-1} = \mu_q = 1/2$. The remaining α_j's can be used to give the method optimal stability characteristics. Since our transport problem usually is convection dominated, we decided to optimize the stability boundary along the imaginary axis (for a more detailed discussion on this aspect, we refer to [8]). It is well known (cf. [7], [6]) that, for *odd* values of q, the second-order scheme (3.5) possesses an optimal imaginary stability boundary $\beta_{imag} = q-1$. In Table 3.1, the corresponding α_j-values are listed for various values of q (for the sake of completeness, we also give the values for β_{real}, the corresponding stability boundary along the real axis). Finally, the requirement of minimal costs is obtained by freezing the t-argument in the first $q-1$ stages, i.e., $\mu_j = 0, j = 1,\ldots,q-1$. In the case of (3.4), a substantial part of the effort in calculating $F(t,C)$ comes from the evaluation of the matrices $A_x(t)$, $A_y(t)$, $A_z(t)$, the vector $G(t,C(t))$, and the evaluation of the boundary conditions. Hence, except for the first and the last stage, these quantities do not need to be evaluated.

Moreover, the storage requirements are relatively modest and the structure of the system (3.4) allows an extremely efficient implementation on vector computers.

Table 3.1. Parameters α_j to obtain an optimal imaginary stability boundary β_{imag} for (3.5)

q	α_1	α_2	α_3	α_4	α_5	α_6	α_7	β_{imag}	β_{real}
4	1/4	1/3						$2\sqrt{2}$	2.78
5	1/4	1/6	3/8					4	2.59
7	1/6	1/12	2/9	4/19	19/54			6	3.00
9	1/8	1/20	5/32	2/17	17/80	5/22	11/32	8	3.31

3.2.2. Odd-Even Line Hopscotch methods

The class of hopscotch methods [5] belongs to the class of so-called operator splitting methods. To define the specific Odd-Even Line Hopscotch (OELH) method that we have in mind, the right-hand side function F in (3.4) is assumed to be split into

$$(3.6)\qquad F(t,C(t)) = F_0(t,C(t)) + F_*(t,C(t)),$$

where F_0 is the vector that is obtained from F by replacing all components corresponding to a grid point $P_{j,k,m}$ where $j+k$ assumes an *even* value by zero. Similarly, F_* is obtained if all elements in F corresponding to a grid point for which $j+k$ is *odd*, are replaced by zero. Notice that the third index, m, is not involved in this definition, which means that we apply the same splitting on each horizontal plane of the grid, or equivalently, *all* grid points lying on the *same* vertical grid line are either in F_0 or in F_*. This observation is crucial for the OELH algorithm.

Now we define the two-stage, second-order splitting method

$$(3.7)\qquad \begin{aligned} C_{n+1/2} &= C_n + \tfrac{1}{2}\Delta t F_0(t_{n+1/2}, C_{n+1/2}) + \tfrac{1}{2}\Delta t F_*(t_n, C_n), \\ C_{n+1} &= C_{n+1/2} + \tfrac{1}{2}\Delta t F_0(t_{n+1/2}, C_{n+1/2}) + \tfrac{1}{2}\Delta t F_*(t_{n+1}, C_{n+1}) . \end{aligned}$$

Owing to the aforementioned splitting, and observing that the discrete system (3.3) possesses a *three*-point coupling in the horizontal direction, we see that - in the first stage - the '$*$-components' of $\boldsymbol{C}_{n+1/2}$ can be computed without evaluating $\boldsymbol{F}_{\text{o}}$. Hence, the first stage can be split into a Forward Euler-type calculation for the '$*$-components', followed by a Backward Euler-type calculation for the 'o-components'. Since the '$*$-components' are already known at the intermediate time level $t_n+\Delta t/2$, each 'o-component' is only coupled in vertical direction. Since we also used a *three*-point coupling to discretize $\partial/\partial z$ and $\partial^2/\partial z^2$, this implicit part of the algorithm results in the solution of tridiagonal systems along each vertical grid line. Of course, similar arguments hold for the second stage, but now the role of both type of components is interchanged.

Using these observations, the scheme (3.7) can be recast in the form

$$
\begin{aligned}
\boldsymbol{C}_{*,n+1/2} &= \boldsymbol{C}_{*,n} + \frac{1}{2}\Delta t\boldsymbol{F}_{*}(t_n,\boldsymbol{C}_n) = \boldsymbol{C}_{*,n} + \left(\boldsymbol{C}_{*,n} - \boldsymbol{C}_{*,n-1/2}\right),\\
\boldsymbol{C}_{\text{o},n+1/2} &= \boldsymbol{C}_{\text{o},n} + \frac{1}{2}\Delta t\boldsymbol{F}_{\text{o}}(t_{n+1/2},\boldsymbol{C}_{n+1/2}),\\
\boldsymbol{C}_{\text{o},n+1} &= \boldsymbol{C}_{\text{o},n+1/2} + \frac{1}{2}\Delta t\boldsymbol{F}_{\text{o}}(t_{n+1/2},\boldsymbol{C}_{n+1/2}) = \boldsymbol{C}_{\text{o},n+1/2} + \left(\boldsymbol{C}_{\text{o},n+1/2} - \boldsymbol{C}_{\text{o},n}\right),\\
\boldsymbol{C}_{*,n+1} &= \boldsymbol{C}_{*,n+1/2} + \frac{1}{2}\Delta t\boldsymbol{F}_{*}(t_{n+1},\boldsymbol{C}_{n+1}).
\end{aligned}
\tag{3.8}
$$

Observe that in the explicit part of both stages the corresponding F-evaluation can be saved. Apart from the very first step, these F-evaluations can be expressed in terms of the concentrations at a previous time level. On using this substitution, we arrive at the so-called 'fast form'. As a result, the total amount of work per step consists in solving $J{\cdot}K$ uncoupled, implicit relations, each of dimension M. For this purpose, one usually applies a (modified) Newton process. Notice that one Newton iteration will suffice if the fall velocity term w_f is not needed in the model, since then the equation is linear in $\boldsymbol{C}$. This case has been considered in the numerical experiments, so that (3.8) requires *one* $\boldsymbol{F}$-evaluation over the whole field, plus the solution of $J{\cdot}K$ tridiagonal linear systems of dimension M (in Section 4, we give a detailed description how these systems can be solved efficiently on a vector computer). It is worth mentioning that the storage requirements of the OELH method are quite modest: apart from the arrays to store the velocity field (which are anyhow needed by every 'transport-solver'), (3.8) needs 2 arrays to store the concentration vector $\boldsymbol{C}$ at the different time levels, 1 array to hold the time derivatives (i.e., DCDT), 3 arrays for housing the tridiagonal Jacobian matrices, and 'half' an array to gather the right-hand side vectors in the linear equations, which amounts to 6.5 arrays (of dimension N).

The main reason to construct this particular variant within the hopscotch-family is that usually the most severe restrictions on the timestep originate from the discretization in the *vertical* direction; that is (cf. (3.3a)), the terms $|w|/\Delta z$ and $\varepsilon_z/(\Delta z)^2$ are usually larger than the similar expressions corresponding to the horizontal direction. Hence, in explicit methods, the timestep will be dictated by stability conditions induced by the vertical discretization. Since the OELH method is explicit in the horizontal, but implicit (and hence unconditionally stable) in the vertical, it usually allows for timesteps which are in accordance with the size that we would require for accuracy reasons.

4. Solving the linear systems in vector mode

In the previous section we have seen that the OELH method (3.8) requires the solution of $J{\cdot}K$ tridiagonal systems of dimension M. In this section we describe the selected linear systems solver which is tailormade for the present application of a three-dimensional numerical transport model. Details of the implementation with respect to optimal vector performance are discussed, as well as the possibility to exploit several parallel processors. Furthermore, we explain some additional options that have been incorporated in the linear solver (which can be of use in the case of different time integrators) and, finally, we present performance results.

4.1. Algorithmic aspects

The decomposition method chosen is $LD^{-1}U$. During development we have compared this decomposition with the obvious alternatives LU and $L^{-1}U$, in particular with respect to overall number of operations and possibilities for fine tuning regarding vector performance. Apart from the opportunity that $LD^{-1}U$ offers to save on the number of divisions, the merits of all methods do not differ conspicuously. The observed slight advantage of the first method can be attributed to the circumstance that it combines best (on average) with all additional facilities that we have chosen to implement, viz., to combine decomposition and solution in one subroutine and the possibility to reuse a stored decomposition. Furthermore, the implementation is suitable for all possible couplings of the unknowns which may stem from a coupling in either x-, y- or z-direction of the physical domain. This feature is not actually exploited by the OELH method, but it gives the linear solver the flexibility to be used in, for example, LOD-type integration methods (see [8] for a discussion of such methods in the present context).

Standard solvers for large tridiagonal systems are publicly available through, e.g., LAPACK [1] (which provides the subroutines SGTTRF for factorizing a tridiagonal matrix and SGTTRS for the subsequent forward/backward substitution) and the Cray Scientific Subroutine library SCILIB (SDTSOL would be a candidate). The main reason for developing our own implementation is the fact that all tridiagonal systems are *uncoupled*, and moreover are of similar shape and have equal dimension. These properties allow an efficient implementation especially for vector processors, employing the method of *vectorization across the systems* which is described below.

We will first describe the $L D^{-1} U$ decomposition method for a *single* tridiagonal system(of dimension n): with proper subscript notation (D is a diagonal matrix, L and U are, respectively, lower and upper bidiagonal matrices, both with unit diagonal) the method reads:

```
$D_1 := 1 / A_{1,1}$
**for** i = 2 **until** n
    $U_{i-1,i} := A_{i-1,i} * D_{i-1}$        ($U_{i-1,i}$ not needed)
    $L_{i,i-1} := A_{i,i-1} * D_{i-1}$
    $D_i := 1 / (A_{i,i} - L_{i,i-1} * A_{i-1,i})$
(Solution of  $A x = b$:)
$x_1 := b_1$
**for** i = 2 **until** n
    $x_i := b_i - L_{i,i-1} * x_{i-1}$
$x_n := D_n * x_n$
**for** i = n−1 **step** −1 **until** 1
    $x_i := D_i * (x_i - A_{i,i+1} * x_{i+1})$
```

An obvious Fortran implementation, which receives the (co)diagonals of the input matrix A in 1-dimensional arrays L(2:n), D(1:n) and U(2:n) and delivers the decomposition results in the same three arrays and the solution in the array B that initially contained the right-hand side vector, is given by:

```
C     Input: matrix in L(2:N), D(1:N), U(2:N), right-hand side in B(1:N).
C     i-th row of L and i-th column of U and i-th diagonal element of D:
      D(1) = 1.0 / D(1)
      DO I = 2, N
         L(I) = L(I) * D(I-1)
         D(I) = 1.0 / (D(I) - L(I) * U(I))
      END DO
C     We now have matrix L with unit diagonal in L, D-1 in D, U is not computed.
C     We proceed with forward, scale and back solve:
      DO I = 2, N
         B(I) = B(I) - L(I) * B(I-1)
      END DO
      B(N) = D(N) * B(N)
      DO I = N-1, 1, -1
         B(I) = D(I) * (B(I) - U(I+1) * B(I+1))
      END DO
C     Solution in B.
```

For the OELH method, in which we have to solve $J{\cdot}K$ systems of dimension M, a straightforward implementation would be to put the above implementation for solving a single system into a loop:

```
DO IND = 1, J*K
   { the previous source code, with N replaced by the actual dimension M of each system }
   { and, since we are dealing with three-dimensional arrays, all subscripts (I) replaced by (IND,1,I) }
   { and similar substitutions for I+1and I-1 }
END DO
```

However, due to the *recursive* nature in the innerloops, this approach will result in a poor performance on vector processors. The trick to obtain good vector code, as described by Golub and Van Loan [4] and called '*vectorization across* *tridiagonal systems*', is to interchange the inner and outer loops. This results in a number of nested loops with the 'IND-loop' as the inner one. E.g., the following loop in the single system implementation

```
DO I = 2, N
  B(I) = B(I) - L(I) * B(I-1)
END DO
```

becomes

```
DO I = 2, N
   DO IND = 1, J*K
      B(IND,1,I) = B(IND,1,I) - L(IND,1,I) * B(IND,1,I-1)
   END DO
END DO
```

In fact, each step in the original Gaussian elimination process is now performed in vector mode, since the vectorizable inner loop runs over all systems. This approach works perfectly well for the present class of problems.

We remark that in the above description the index `I` is always the *third* index in the three-dimensional arrays; this is because the OELH method is implicit in the vertical (i.e., z-) direction, which corresponds with the third index. As a matter of fact, our tridiagonal solver is more general in the sense that it is also capable of treating systems originating from implicitness in x- or y-direction; in those cases, the index `I` will, respectively, be the first and second index in all three-dimensional arrays. In the next subsection, the influence on the performance is shown. For full details on all three cases of coupling, we refer to the Fortran-listing in the Appendix (subroutine TRI3S5 and its auxiliaries TRI3P1 and TRI3P3).

4.2. Implementational aspects

We will briefly touch a number of issues that are of importance for the development of the final implementation. During the development several design decisions concerning details were taken after intensive measuring of performance differences for all of the possible branches discussed below.

- *Branching with respect to the direction of the coupling of unknowns*:
  Different implementations are used to treat the linear systems arising from a coupling in the x-, y- or z-direction of the physical domain. In these cases the order in which data storage places are accessed is completely different.
- *Keeping and reusing the decomposition results*:
  One design decision taken is to store the decomposition results in such a way that they can be reused efficiently. Therefore, the results of the decomposition (and of the solution) are delivered in the storage space containing the original input. Although costs are connected with this way of storing the decomposition, the advantage is clear in case the decomposition is reused for a system with only a new right-hand side.
- *Parallel processing possibility*:
  The final implementation of the tridiagonal systems solver is parametrized with *NPP*, the number of parallel processors to be employed. It is extremely simple to split the complete workload in *NPP* portions of (approximately) equal size, which are completely independent. A speedup of 4 (using all 4 processors of the Cray Y-MP4/464) would be obtainable, but the splitting of long loops into 4 parts slightly decreases the performance on a single vector processor. This is confirmed by a couple of experiments where we obtained a parallel speedup of 3.3 - 3.8.
- *Fine tuning*:
  The actual implementation receives the data of a Jacobian matrix J and a time increment Δt from which it first computes the coefficient matrices $A = I - \Delta t\, J$ that will subsequently be decomposed. Much attention has been given to combining loops that are described as different tasks into single loops whenever possible. This loop combining has been applied for the above mentioned setting up of A and its decomposition, but also for combining decomposition and forward substitution.
- *About vectorization across tridiagonal systems*:
  In order to employ vectorization across tridiagonal systems Golub and Van Loan recommended that the data should be reordered such that successive computations (i.e. corresponding iterations of different tridiagonal systems) can access successive array elements. In our application this would require complete copying of the data in a different order before proceeding with the decomposition. However, on the CRAY Y-MP4/464 such a data conversion is not at all needed as processing is equally efficient with loops in which we jump with large but constant stride through the accessed memory.

In Table 4.1 we summarize the results of timing experiments for the tridiagonal systems solver. The experiments have been carried out with $J = K = 101$ and $M = 11$, hence all arrays have dimensions: (101, 101, 11).

Table 4.1. CPU times (in milli-seconds) and Mflop rates on a CRAY Y-MP4/464, using 1 processor

	scalar mode		vector mode		
coupling	CPU	Mflops	CPU	Mflops	speedup
in x-direction	108.0	16.6	9.4	189.1	11.4
in y-direction	105.0	17.1	9.2	194.2	11.4
in z-direction	96.7	17.6	8.3	205.9	11.7

5. Numerical experiments

To show the performance of the methods described in the previous sections, we take the following example problem (see also [8])

$$\text{(5.1a)}\qquad \frac{\partial c}{\partial t} + \frac{\partial (uc)}{\partial x} + \frac{\partial (vc)}{\partial y} + \frac{\partial (wc)}{\partial z} = \varepsilon \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) c + g(t,x,y,z), \quad 0 \le t \le T\,,$$

with Neumann conditions of the form $\partial c/\partial n = h(t,x,y,z)$ on all boundaries of the physical domain $0 \le x \le L_x$, $0 \le y \le L_y$, $-L_z \le z \le 0$. In our experiments, it is convenient to have the exact solution at our disposal. In terms of the scaled coordinates $\tilde{x} := x/L_x$, $\tilde{y} := y/L_y$, $\tilde{z} := z/L_z$, the concentration c is prescribed as

$$\text{(5.1b)}\qquad c(t,x,y,z) = \exp \left\{ \tilde{z} - f(t) - \gamma \left[(\tilde{x} - r(t))^2 + (\tilde{y} - s(t))^2 \right] \right\},$$

where the functions r, s and f are defined by $r(t) := [2 + \cos(2\pi t/T_p)]/4$, $s(t) := [2 + \sin(2\pi t/T_p)]/4$, and $f(t) := 4t/(T_b+t)$. Hence, a local concentration is advected in a rotation (with period T_p) around the centre of the domain. Furthermore, we see that the concentration is slightly decreasing in the vertical direction and damped in time (to get a better impression of the behaviour of this function, we refer to the pictures in the Figures 5.1 where the numerical solution is plotted for various points in time). The inhomogeneous function g and the function h defining the boundary conditions follow from this exact solution. Owing to the special (i.e., exponential) structure of (5.1b), the functions g and h can be written in the form $g = c\,\tilde{g}$ and $h = c\,\tilde{h}$. This form is more convenient in the implementation.

We take the following values for the parameters: $L_x = L_y = 20\,000$ [m], $L_z = 100$ [m], $\varepsilon = 0.5$ [m^2/s], $T_p = 43200$ [s] (12 hours), and $T_b = 32400$ [s]. The (dimensionless) parameter γ serves to adjust the *local* nature of the concentration and has been set to 10. For the length of the integration interval T we have used two different values, viz. $T_1 = 10\,800$ [s] (3 hours) and $T_2 = 432\,000$ [s] (which equals 5 days). The particular value will be specified in the tables of results (at these points in time, the damping effect is $\exp(-f(T_1)) \approx 0.37$ and $\exp(-f(T_2)) \approx 0.024$). The concentration c is assumed to be measured in kg/m^3.

The velocity fields are prescribed by the analytical expressions

$$\text{(5.2)}\qquad \begin{aligned} u(t,x,y,z) &= C_1 \sin(\tilde{x} + \tilde{y}) \sin(\beta\,\tilde{z})\, d(t), \\ v(t,x,y,z) &= C_2 \cos(\tilde{x} + \tilde{y}) \sin(\beta\,\tilde{z})\, d(t), \\ w(t,x,y,z) &= \left[-\frac{C_1}{L_x} \cos(\tilde{x} + \tilde{y}) + \frac{C_2}{L_y} \sin(\tilde{x} + \tilde{y}) \right] \left[-\frac{L_z}{\beta} \cos(\beta\,\tilde{z}) \right] d(t), \end{aligned}$$

with $d(t) = \cos(2\pi t/T_p)$, and $C_1 = 3$, $C_2 = 4$, $\beta = 0.05$. We remark that these fluid velocities satisfy the relation for local mass balance, i.e.,

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0,$$

so that (5.1a) can be simplified indeed to the form (3.1); the fall velocity term w_f is not taken into account in our tests.

To obtain insight to what extent the length of the vectors will influence the performance of the algorithms, we have done experiments on two spatial grids:

Grid I is defined by: $J = K = 101$, $M = 11$, amounting to $\approx 10^5$ grid points;
Grid II is defined by: $J = K = 201$, $M = 21$, amounting to $\approx 8.5\ 10^5$ grid points.

5.1. Algorithmic tests of the time integration methods

First, we will compare the time integrators with respect to their *numerical* properties, like accuracy and stability. Therefore, they are applied to problem {(5.1),(5.2)}, defined on Grid I. In the next table we show the (numerical) behaviour of the OELH method and several stabilized RK methods, based on q stages (in the sequel, these methods will be denoted by RK2q). As the end point of the integration we choose $T_1 = 10\,800$.

Table 5.1. Global errors for OELH and several stabilized RK methods at $T_1 = 10\,800$

# steps	Δt	OELH	RK24	RK25	RK27	RK29
5	2160	0.0066	*	*	*	*
10	1080	0.0016	*	*	*	*
20	540	0.00075	*	*	*	*
40	270	0.00055	*	*	*	*
80	135	0.00051	*	*	*	0.00050
95	114	0.00051	*	*	0.00050	0.00050
125	86.4	0.00051	*	0.00050	0.00050	0.00050
160	67.5	0.00050	0.00050	0.00050	0.00050	0.00050

This table shows that the OELH is the most stable time integration process (an * denotes unstable behaviour). Moreover, it is quite accurate, since already for Δt as large as 270, the time integration error is almost negligible compared to the spatial discretization error, which is seen to be 0.0005 for this grid. Furthermore, we observe that the stability of the RK methods is improved as the number of stages increases.

The next question is of course, which stabilized RK method is the most efficient one, i.e., what is the optimal q-value. To answer this question, we have to take into account the costs of the various stages. As pointed out in Section 3.2.1, an RK2q method requires 2 'expensive' F-evaluations and q–2 'cheap' F-evaluations per step (since the overhead in an RK method is negligible, we only count F-evaluations). Let σ denote the fraction that can be saved by evaluating a 'cheap' F. Then the total costs $E(q,\sigma)$ (in terms of full F-evaluations) over the whole integration interval is given by $E(q,\sigma) := N_{st}[2+(q-2)(1-\sigma)]$, where N_{st} is the minimal number of timesteps that has to taken for stability reasons. If N_{st} is only determined by β_{imag}, then $E(q,\sigma)$ reduces to $D[q(1-\sigma)+2\sigma)]/(q-1)$, where D is a constant depending on the length of the integration interval and on the spectral radius of $\partial F/\partial C$. In this case $E(q,\sigma)$ is a monotonically decreasing function of q, which suggests a large q-value (yielding increasing efficiency as $\sigma \to 1$). However, in practice, also the value of β_{real} is relevant, especially if $\Delta z \to 0$ (see also the experiments described in Section 5.3). This is already noticeable from the maximally allowed time steps as listed in Table 5.1, which show that the increase in Δt is less than might be expected on the basis of β_{imag}. In conclusion, the optimal value of q depends on the ratio between $|w|/\Delta z$ and $\varepsilon_z/(\Delta z)^2$, and on the value of σ. For the problem described in this section, $q = 7$ turned out to be a good choice. Therefore, in the sequel, we will confine ourselves to the OELH and the RK27 method.

Next, we let the methods be subject to a more severe stability test by integrating upto $T_2 = 432\,000$. The results, given in Table 5.2, reveal that both methods show a stability behaviour that is similar to their behaviour on the short integration interval. Furthermore, in the accuracy region where the temporal error is dominating the spatial error, the OELH method shows its second-order behaviour.

Table 5.2. Global errors for OELH and RK27 at $T_2 = 432\,000$

# steps	Δt	OELH	RK27
200	2160	*	*
230	1878	0.0097	*
400	1080	0.0044	*
800	540	0.0020	*
1600	270	0.0014	*
3200	135	0.0013	*
3500	123	0.0013	*
3800	114	0.0013	0.0013
4000	108	0.0013	0.0013

The Figures 5.1 show the initial concentration $c(0,x,y,z)$ according to (5.1b), and the numerical solutions at $t = T_1$ and $t = T_2$ obtained with the OELH method using $\Delta t = 270$.

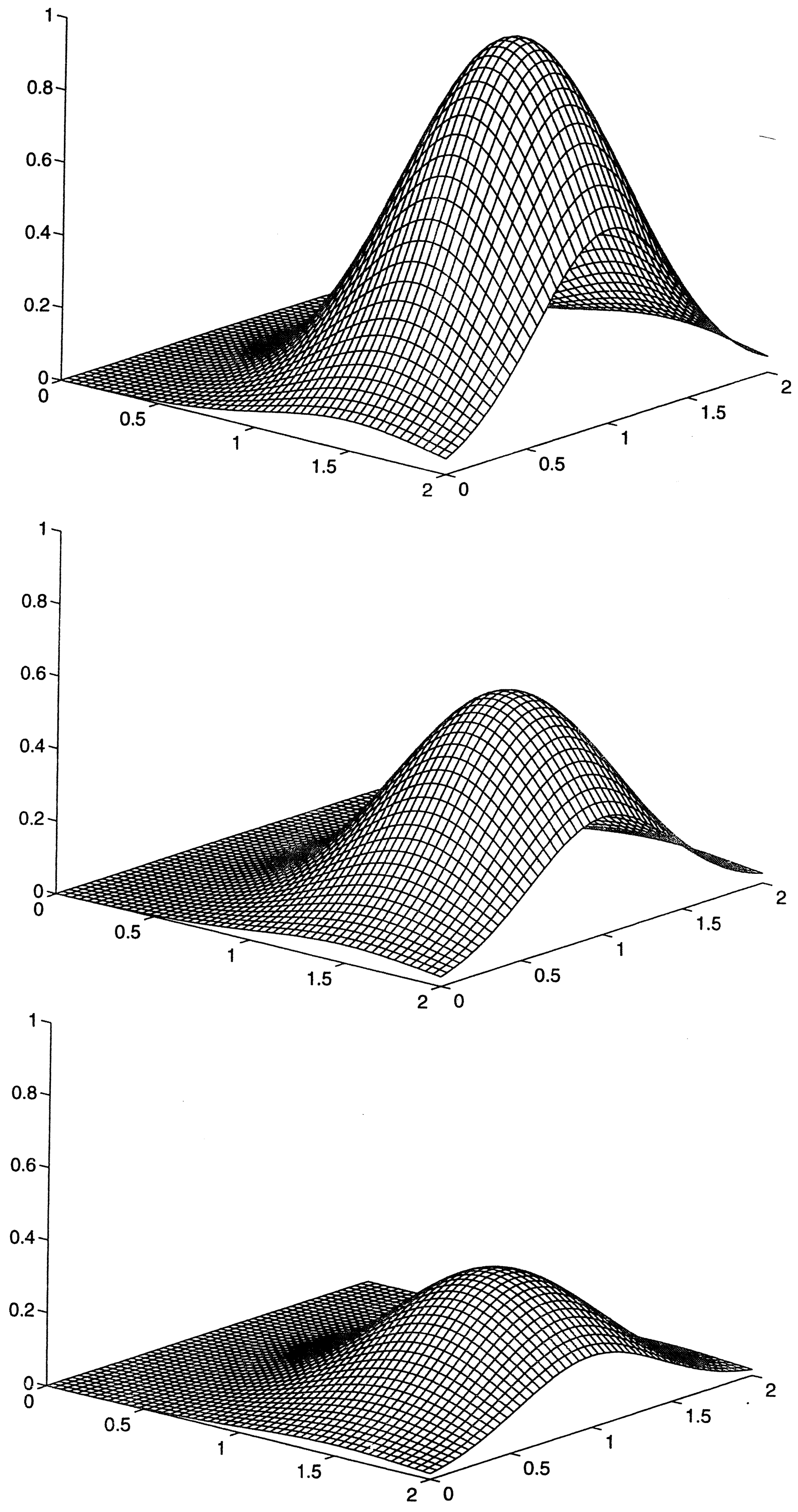

Figure 5.1a. The concentration $c(t,x,y,z)$ for $t = 0$ at three horizontal planes, viz. at the surface ($z = 0$), halfway down ($z = -50$m) and at the bottom ($z = -100$m)

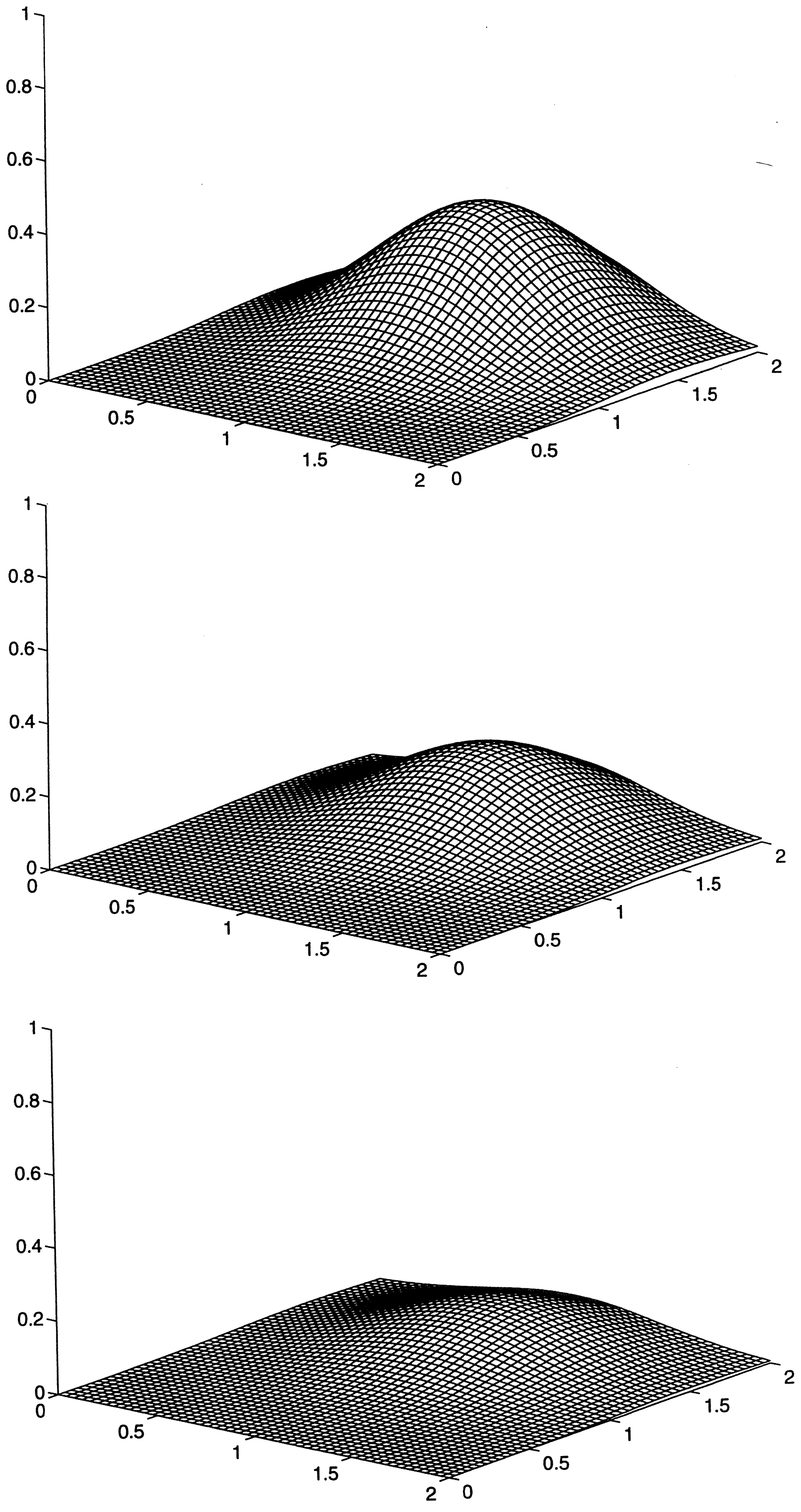

Figure 5.1b. The computed concentration after 3 hours ($t = T_1 = 10\,800$) at three horizontal planes, viz. at the surface ($z = 0$), halfway down ($z = -50$m) and at the bottom ($z = -100$m). At the surface, the maximum of the concentration equals 0.37. Furthermore, we see that the solution has been advection over 90 degrees.

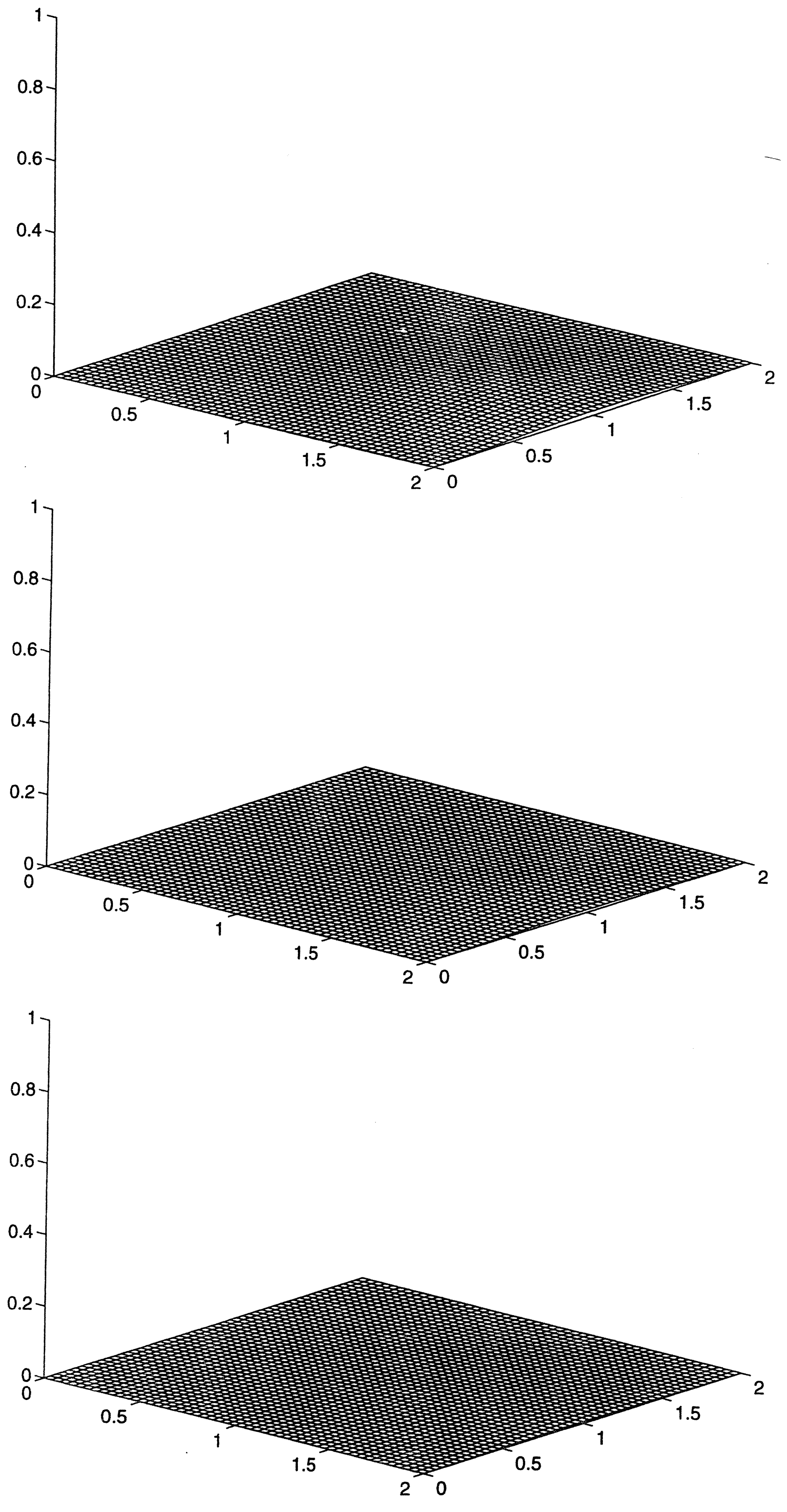

Figure 5.1c. The computed concentration after 5 days ($t = T_2 = 432\,000$) at three horizontal planes, viz. at the surface ($z = 0$), halfway down ($z = -50$m) and at the bottom ($z = -100$m).
At the surface, the maximum of the concentration is now reduced to 0.0228.

5.2. Performance results

In this section we describe the performance of both integration methods when implemented on the CRAY Y-MP 4/464. In all experiments we used the CF77 compiling system. Performance results in scalar and vector mode are respectively obtained using **cf77 -Wf" -o novector"** and **cf77 -Zv**. Moreover, we frequently used the package **perfview** [2] (using the flags **-F** and **-lperf**) to obtain the Megaflop rates and CPU times of the various routines. All experiments have been performed on 1 vector pipe (unless otherwise stated). Since the clock cycle time of the CRAY Y-MP 4/464 equals 6 nanoseconds, the optimal vector speed equals 167 Mflops; in the exceptional case that a multiply and an addition can always be chained, this theoretical peak performance can be multiplied by a factor 2.

In Table 5.3 the performance results (on Grid I and end point $T_1 = 10\,800$) are given for the RK27 method, using the smallest number of steps, i.e., $N_{st} = 95$.

Table 5.3. Vector performance of the main routines in RK27

routine	number of calls	average time (in seconds)	accumulated percentage of CPU-time spent	Mflop rate
F (derivative)	665	1.5 E−2	76.2	212
G (source term)	190	1.2 E−2	93.3	197
RK (driver)	1	6.9 E−1	98.7	266

The performance results for the OELH method are given in Table 5.4. Here, we used $N_{st} = 40$, corresponding to $\Delta t = 270$. This timestep yields approximately the same global error as the RK27 method using the largest possible step.

Table 5.4. Vector performance of the main routines in OELH

routine	number of calls	average time (in seconds)	accumulated percentage of CPU-time spent	Mflop rate
F (derivative)	81	9.1 E−3	33.7	169
G (source term)	81	6.6 E−3	58.2	178
TRI3P3 (tridiagonal solver)	80	4.2 E−3	73.5	205
JACOB (evaluates Jacobian)	80	2.3 E−3	81.7	116
IMPL (solves implicit relation)	80	2.0 E−3	89.0	112
HS (driver)	1	8.1 E−2	92.7	223

These tables give rise to the following remarks and conclusions:

- The high Megaflop rates obtained in most routines clearly indicate that both time integrators vectorize extremely well. Since these methods have attractive numerical properties as well, they both are candidates for the efficient solution of transport problems.
- Except for the driver, most routines in OELH are called about $2N_{st}$ times; the factor 2 originates from the fact that the OELH method has 2 stages (notice that the fast form of OELH requires some initialization, resulting in one extra call of F and G). Observe that the average time spent in F is much lower than the corresponding time for RK27. This is because OELH needs to evaluate (per call of F) only half the number of points. The stride 2 that is involved in the hopscotch algorithm slightly reduces the vector performance. The same arguments apply to G. This stride effect is more pronounced in the routines IMPL and JACOB, since there the number of arithmetic operations is relatively low.
- It should be remarked that in our test problem, the g-function is unrealistically expensive. This is because we have used this function to let the prescribed concentration (5.1b) be the exact solution. In real-life problems this function will be much cheaper. If we ignore the time that the integrators spent in the routine G, then RK27 spends 92.0 % of its time in the derivative function; for OELH this change would result in 44.7 % for F. For a code based on an *implicit* method, this percentage is pretty high, showing that the linear algebra part is efficiently treated by the vectorization-across-the-tridiagonal-systems approach. It is of interest to mention that if this part would have been performed in the traditional (i.e., recursive, and hence non-vectorizable) way, then the first three lines in Table 5.4 would change to:

routine	number of calls	average time (in seconds)	accumulated percentage of CPU-time spent	Mflop rate
TRI3P3 (tridiagonal solver)	80	4.9 E−2	68.1	17
F (derivative)	81	9.1 E−3	80.8	169
G (source term)	81	6.6 E−3	90.0	178

This is in accordance with the usual experience that an implicit method spends most of its time in solving the systems.

Of course, more interesting than the vector performance of the algorithms is the actual CPU time that they need to arrive at an accurate solution. A global performance is given in the following table, where we have given the performance in scalar mode as well.

Table 5.5. Global performance of OELH and RK27 on Grid I

	OELH ($N_{st}=40$)		RK27 ($N_{st}=95$)	
	CPU (sec.)	Mflop rate	CPU (sec.)	Mflop rate
scalar mode	15.6	24	99.2	28
vector mode	2.2	170	12.9	211

From this table we see that the methods are 7-8 times faster when run in vector mode; this factor confirms the excellent vectorization capabilities of both algorithms. Furthermore, it is clear that the OELH method is able to produce an accurate result in roughly 20% of the time needed by RK27. Although this last method shows a very high Megaflop rate, its timestep restriction makes it less efficient than OELH.

5.3. Influence of the spatial grid

Next, we solve the problem {(5.1),(5.2)}, discretized on Grid II. It is to be expected that the refinement of the spatial grid will have influence on the accuracy, on the stability, and on the vector performance. For the end point of the integration interval we choose T_1. The results are listed in Table 5.6.

Table 5.6. Global errors (on Grid II) for OELH and RK27 at $T_1 = 10\,800$

# steps	Δt	OELH	RK27
10	1080	0.00242	*
20	540	0.00063	*
40	270	0.00019	*
80	135	0.00013	*
160	67.5	0.00013	*
280	38.6	0.00013	*
290	37.2	0.00013	0.00013
320	33.75	0.00013	0.00013

For $\Delta t \to 0$, the error in both methods converges to 0.00013, which is the spatial discretization error on this fine grid. This value is four times smaller than the error obtained on Grid I (cf. Table 5.1), which is in agreement with the second-order spatial discretization that we have used.

With respect to stability we may conclude that the OELH method behaves as stable as it did on the 'coarse' grid. However, due to the increased stiffness of the resulting ODE, the RK27 method is forced to take timesteps that are reduced by a factor 3. This factor indicates that the terms originating from the diffusion part and from the advection part in the PDE are *both* of influence on the maximal timestep: if one of these terms would have been dominating, then one would obtain a reduction of the timestep by a factor 4 or 2, respectively.

It is also of interest to see the influence of the grid refinement (i.e., longer vectors) on the performance. The Tables 5.7 and 5.8 give this information for RK27 (with $N_{st}=290$) and OELH (using $N_{st}=40$), respectively. Comparing the Tables 5.3 and 5.7, we see that the performance of RK27 is hardly improved: the 'coarse' Grid I is sufficiently fine to let this method run at a speed close to the optimal speed. The overall performance of this method increases from 211 to 219 Mflops. A comparison of the Tables 5.4 and 5.8 shows that the average Mflop rate of the OELH method is increased by approximately 10 % (the overall performance changes from 170 to 184 Mflops).

Table 5.7. Vector performance of the main routines in RK27 (Grid II, $N_{st}=290$)

routine	number of calls	average time (in seconds)	accumulated percentage of CPU-time spent	Mflop rate
F (derivative)	2030	1.1 E−1	77.4	218
G (source term)	580	8.2 E−2	94.3	210
RK (driver)	1	1.4 E+1	99.5	267

Table 5.8. Vector performance of the main routines in OELH (Grid II, $N_{st} = 40$)

routine	number of calls	average time (in seconds)	accumulated percentage of CPU-time spent	Mflop rate
F (derivative)	81	6.3 E–2	34.0	183
G (source term)	81	4.3 E–2	57.7	200
TRI3P3 (tridiagonal solver)	80	3.2 E–2	74.4	206
JACOB (evaluates Jacobian)	80	1.3 E–2	81.9	138
IMPL (solves implicit relation)	80	1.3 E–2	89.3	128
HS (driver)	1	6.3 E–1	93.0	220

5.4. Parallelization aspects

Finally, we consider the prospects that the methods offer with respect to parallelization. In both codes we frequently encounter the situation that we have a 3-times nested loop of the following structure:

```
      DO 10 k = 1, K   (in horizontal space direction)
      DO 10 m = 1, M   (in vertical space direction)
      DO 10 j = 1, J   (in horizontal space direction)
         ...
         body of the loop
         ...
   10 CONTINUE
```

A typical treatment of such loops on parallel/vector machines is to vectorize the inner loop (index j), and to parallelize the outer loop (index k), whereas the loop with index m is performed in sequential mode (or, if possible, collapsed with the j-loop).

We have used the autotasking facility of the CRAY (specifying **cf77 -Zp** activates the Fortran-preprocessor FPP to analyse the code) to get an indication of the speed-up that can be obtained for OELH when various vector pipes are used. To that end we employed the so-called **atexpert** utility [2]. Atexpert is capable of producing predictions of the performance on a dedicated system. The results obtained by this utility, when running OELH on the coarse grid, are given in Table 5.9. The speed-up factors clearly indicate that the OELH method efficiently exploits a multiprocessor machine like the CRAY.

Table 5.9. Parallel performance of OELH, estimated by atexpert

# processors:	1	2	3	4	5	6	7	8
speed-up:	1.00	1.99	2.93	3.82	4.71	5.54	6.35	7.13

6. Concluding remarks

In this paper we have discussed stabilized Runge-Kutta (RK) methods and the Odd-Even Line Hopscotch (OELH) method. From a previous evaluation study [8], both types of methods turned out to be suitable candidates for the time integration of a three-dimensional numerical transport model. These methods have quite different numerical properties: the RK methods are explicit, and hence they have to obey a rather restrictive stepsize condition; since the OELH method is partly implicit, this method is, in many practical situations, not hampered by such a stability-induced stepsize restriction. Both methods also differ with respect to vectorization capabilities: because of the extremely simple nature of the RK methods, they are straightforwardly implemented on a vector processor machine. On the basis of large scale test problems ($\approx 10^5$ - 10^6 unknowns), we measured a vector performance over 200 Mflops on a 1-processor CRAY Y-MP4/464 (having a clock cycle of 6 nanoseconds). On first sight, the OELH method seems to be less suitable for vectorization, since it has to solve tridiagonal systems. However, since all these systems are uncoupled, they can be efficiently implemented on vector machines. Following this approach, which was initially proposed by Golub and Van Loan [4], the overall performance of the OELH method turns out to be 170-184 Mflops. Of course, these high Mflop rates only show that the methods are well suited for vectorization; a good numerical solver should, in addition, have proper algorithmic characteristics, resulting in the ultimate goal: small CPU times. Comparing both methods in this respect, we see that the OELH method is by far superior to the RK methods. The fact that OELH can take much larger timesteps easily compensates for the (slightly) lower vector speed. Furthermore, we have seen that OELH is also capable of exploiting the parallel facilities offered by a multiprocessor architecture.

As explained in Section 3, we have chosen for the so-called 'dummy-point' approach, which means that the physical domain is enclosed by a rectangular box. In the test problem discussed in Section 5, this approach did not lead to dummy points, so that all floating point operations were really useful operations. In practical situations however, where we have capricious geometries (e.g., in estuaries, coasts and bottom profiles), the situation is less ideal and the introduction of dummy points is unavoidable. In such cases, we should consider an *effective* Mflop rate, which obviously has to be related to the number of *useful* (i.e., effective) floating point operations. However, this observation applies to any integration method since this 'dummy-point' approach is inherent to the spatial discretization and the choice of the data structures.

Summarizing, we conclude that the OELH method is an efficient method for the integration of three-dimensional transport models because it combines in a suitable way the following properties: sufficient accuracy, sufficient stability, modest storage requirements, almost optimal vectorization and parallelization capabilities, and low computational costs.

Acknowledgement. The authors are grateful to Prof. P.J. van der Houwen for his constructive remarks.

References

[1] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenny, A., and Sorensen, D., *Preliminary LAPACK Users' Guide*, 1991.

[2] Cray Research, *Inc. UNICOS Performance Utilities Reference Manual, SR-2040, edition 6.0.*

[3] Goede, E.D. de, *Numerical methods for the three-dimensional shallow water equations on supercomputers*, Thesis, University of Amsterdam, 1992.

[4] Golub, G.H. and Van Loan, C.F., *Matrix Computations*, The Johns Hopkins Press, Baltimore, Maryland, 2nd edition, 1989.

[5] Gourlay, A.R., *Hopscotch: a fast second order partial differential equation solver*, J. Inst. Math. Appl. 6, 1970, 375-390.

[6] Hairer, E. and Wanner, G., *Solving Ordinary Differential Equations II; Stiff and Differential-Algebraic Problems*, Springer Series in Comput. Math., Vol. 14, Springer, Berlin, 1989.

[7] Houwen, P.J. van der, *Construction of Integration Formulas for Initial Value Problems*, North-Holland series in Applied Mathematics and Mechanics, Vol. 19, North-Holland, 1977.

[8] Sommeijer, B.P., van der Houwen, P.J. and Kok, J., *Time integration of three-dimensional numerical transport models*, Report NM-R9316, CWI, Amsterdam, 1993.

[9] Toro, M., Rijn, L.C. van and Meijer, K., *Three-dimensional modelling of sand and mud transport in currents and waves*, Technical Report No. H461, Delft Hydraulics, Delft, The Netherlands, 1989.

[10] Vreugdenhil, C.B. and Koren, B. (eds.), *Numerical Methods for Advection-Diffusion Problems*, Notes on Numerical Fluid Mechanics, Vol. 45, Vieweg, Braunschweig, 1993.

Appendix

Here we give the Fortran 77 code of the OELH method for solving problem {(5.1),{5.2)}; the grid defining parameters correspond with Grid I. Furthermore, this appendix presents the code of the tridiagonal systems solver, consisting of the shell subroutine TRI3S5, and the core subroutines TRI3P1 and TRI3P3.

```
      program oelh
      parameter(mx=102,my=102,mz=12,
     +          nx=mx-1,ny=my-1,nz=mz-1,nxh=(nx+1)/2)
      dimension c(0:mx,0:my,0:mz),ch(0:mx,0:my,0:mz),
     +          cdot(nx,ny,nz),gtilde(nx,ny,nz),rhs(nxh,ny,nz),
     +          tmodd(nxh,ny,nz),tuodd(nxh,ny,nz),tlodd(nxh,ny,nz),
     +          tmeven(nxh,ny,nz),tueven(nxh,ny,nz),tleven(nxh,ny,nz),
     +          us(nx,ny,nz),vs(nx,ny,nz),ws(nx,ny,nz)
      real lx,ly,lz
      logical newjac,newalp
      common/param/diffx,diffy,diffz
      common/grid/dx,dy,dz
      common/time/tp,tb
      common/domain/lx,ly,lz
      common/coef1/cf1,cf2,beta
      common/coef2/gamma,pi
c---------------------------------------
c   definition of the physical parameters
c---------------------------------------
      diffx=0.5
      diffy=0.5
      diffz=0.5
      lx=2.0e4
      ly=2.0e4
      lz=1.0e2
      cf1=3.0
      cf2=4.0
      beta=5.0e-2
      gamma=10.0
      pi=4.0*atan(1.0)
      tp=43200.0
      tb=32400.0
c--------------------------------
c   define the spatial grid sizes
c--------------------------------
      dx=lx/(nx-1)
      dy=ly/(ny-1)
      dz=lz/(nz-1)
c----------------------------------------------------
c   read the input parameters:
c      the end point of the integration interval,
c      the number of parallel vector pipes to be used, and
c      the number of timesteps
c----------------------------------------------------
      read(5,*)tend
      read(5,*)nproc
      read(5,*)nsteps
      dt=tend/nsteps
      dthalf=dt/2
c-------------------------------------------
c   initialization;
c   the space dependent part of the prescribed
c   flow field is calculated prior to the time
c   integration and stored in the arrays us, vs,
c   and ws (cf. formula (5.2))
c-------------------------------------------
      t=0.0
      call sol(mx,my,mz,nx,ny,nz,t,c)
      call uspace(nx,ny,nz,us)
      call vspace(nx,ny,nz,vs)
      call wspace(nx,ny,nz,ws)
c------------------------------------
c   start of the integration process
c------------------------------------
      call second(t0)
      do 40 ns=1,nsteps
c-----------------
c   first stage
c-----------------
      if(ns.eq.1)then
c-----------------------------------------------------
c   during the first timestep, the explicit part of the
c   first stage needs the calculation of the derivative
c-----------------------------------------------------
         call f(mx,my,mz,nx,ny,nz,t,c,cdot,1,us,vs,ws,gtilde)
         do 10 j=1,ny
         do 10 k=1,nz
         do 10 i=1,nx
            ch(i,j,k)=c(i,j,k)+dthalf*cdot(i,j,k)
   10    continue
         newalp=.true.
      else
c--------------------------------------------------------
c   in subsequent steps, the f-evaluation can be expressed
c   in terms of the concentrations at previous time levels,
c   resulting in a simple extrapolation formula;
c   this is the so-called fast-form (cf. formula (3.8))
c--------------------------------------------------------
         do 20 j=1,ny
         do 20 k=1,nz
         do 20 i=1,nx
            ch(i,j,k)=2.0*c(i,j,k)-ch(i,j,k)
   20    continue
         newalp=.false.
      endif
c------------------------------------------
c   solve the implicit part of the first stage
c------------------------------------------
      call impl(mx,my,mz,nxh,nx,ny,nz,t+dthalf,c,ch,cdot,
     +          tmeven,tueven,tleven,rhs,2,newjac,newalp,
     +          dthalf,us,vs,ws,nproc,gtilde)
c-----------------
c   second stage
c-----------------
      do 30 j=1,ny
      do 30 k=1,nz
      do 30 i=1,nx
         c(i,j,k)=2.0*ch(i,j,k)-c(i,j,k)
   30 continue
c-------------------------------------------
c   solve the implicit part .of the second stage
c-------------------------------------------
      call impl(mx,my,mz,nxh,nx,ny,nz,t+dt,ch,c,cdot,
     +          tmodd,tuodd,tlodd,rhs,1,newjac,newalp,
     +          dthalf,us,vs,ws,nproc,gtilde)
      t=t+dt
   40 continue
      call second(t1)
```

```
c----------------------------------------------------
c   calculation of the global error in the end point
c   (maximum norm)
c----------------------------------------------------
      call sol(mx,my,mz,nx,ny,nz,t,ch)
      errmax=0.0
      do 50 j=1,ny
      do 50 k=1,nz
      do 50 i=1,nx
         err=abs(c(i,j,k)-ch(i,j,k))
         if(err.gt.errmax)then
            errmax=err
            isave=i
            jsave=j
            ksave=k
         endif
   50 continue
      write(6,1)t,errmax,isave,jsave,ksave,t1-t0
    1 format(' at t=',f15.2,' the maximum error=',e12.3,
     +       ' at the point: (',i3,',',i3,',',i3,')',/
     +       ' the integration required',f15.2,' second')
      end


      subroutine impl(mx,my,mz,nxh,nx,ny,nz,t,cold,cnew,dcdt,
     +                tm,tu,tl,rhs,ipar,newjac,newalp,alpha,
     +                us,vs,ws,nproc,gtilde)
c-----------------------------------------------------------------
c   the subroutine impl solves the implicit part of both stages
c   by using a Newton-type method. since our test problem is
c   linear, we applied only one Newton iteration.
c   the parameter ipar determines whether we are dealing
c   with the "odd" or the "even" points
c-----------------------------------------------------------------
      dimension cold(0:mx,0:my,0:mz),cnew(0:mx,0:my,0:mz),
     +          dcdt(nx,ny,nz),gtilde(nx,ny,nz),
     +          tm(nxh,ny,nz),tu(nxh,ny,nz),tl(nxh,ny,nz),
     +          rhs(nxh,ny,nz),
     +          us(nx,ny,nz),vs(nx,ny,nz),ws(nx,ny,nz)
      logical newjac,newalp
c
      call f(mx,my,mz,nx,ny,nz,t,cnew,dcdt,ipar,us,vs,ws,gtilde)
      call jacob(mx,my,mz,nxh,nx,ny,nz,t,cnew,tm,tu,tl,ipar,ws,gtilde)
      newjac=.true.
c--------------------------------------------------------------------
c   calculate the right-hand sides vector for the Newton iteration
c--------------------------------------------------------------------
      do 20 j=1,ny
         do 10 k=1,nz
         ib=mod(j+ipar,2)+1
         lc=0
         do 10 i=ib,nx,2
            lc=lc+1
   10       rhs(lc,j,k)=cnew(i,j,k)-cold(i,j,k)-alpha*dcdt(i,j,k)
   20 continue
c-------------------------
c   solve the linear systems
c-------------------------
      call tri3s5(nproc,tl,tm,tu,rhs,3,nxh,ny,nz,alpha,newjac,
     +            newalp,ifail)
      if(ifail.ne.0)then
         write(6,1)ifail,t
   1    format('  ifail=',i3,'  at t=',f12.4)
        stop
      endif
c------------------------------
c   apply the Newton correction
c------------------------------
      do 40 j=1,ny
         do 30 k=1,nz
         ib=mod(j+ipar,2)+1
         lc=0
         do 30 i=ib,nx,2
            lc=lc+1
  30        cnew(i,j,k)=cnew(i,j,k)-rhs(lc,j,k)
  40  continue
      return
      end


      subroutine f(mx,my,mz,nx,ny,nz,t,c,dcdt,ipar,us,vs,ws,gtilde)
c---------------------------------------------------------------
c   the subroutine f calculates the time derivatives of the
c   concentrations on the computational set of grid points
c   S, as defined in formula (3.2b).
c   the parameter ipar determines whether we are dealing
c   with the "odd" or the "even" points
c---------------------------------------------------------------
      dimension c(0:mx,0:my,0:mz),dcdt(nx,ny,nz),
     +          us(nx,ny,nz),vs(nx,ny,nz),ws(nx,ny,nz),
     +          gtilde(nx,ny,nz)
      common/param/diffx,diffy,diffz
      common/grid/dx,dy,dz
      common/time/tp,tb
      common/coef2/gamma,pi
c---------------------------------------------------------------
c   the time dependent part of the flow field is calculated
c   the source term is stored in the array gtilde
c---------------------------------------------------------------
      d=cos(2.0*pi*t/tp)
      call g(nx,ny,nz,t,gtilde,us,vs,ws,ipar)
c---------------------------------------------------------------
c   next, the (Neumann) boundary conditions are used to
c   fill the auxiliary elements of the array c that
c   correspond with the boundaries of the box P (cf.
c   formula (3.2a))
c---------------------------------------------------------------
      call dcdx(mx,my,mz,nx,ny,nz,t,c)
      call dcdy(mx,my,mz,nx,ny,nz,t,c)
      call dcdz(mx,my,mz,nx,ny,nz,t,c)
c--------------------------------------------
c   initialization of the field of derivatives
c--------------------------------------------
      do 10 j=1,ny
      do 10 k=1,nz
      do 10 i=1,nx
         dcdt(i,j,k)=0.0
  10  continue
c-----------------------------------------------------------------
c   calculate the time derivatives in the grid points defined
c   by ipar (i.e., the "odd" or the "even" points)
c-----------------------------------------------------------------
      do 20 j=1,ny
      do 20 k=1,nz
      ib=mod(j+ipar,2)+1
      do 20 i=ib,nx,2
         dcdt(i,j,k)=
     +      - d*(us(i,j,k)*(c(i+1,j,k)-c(i-1,j,k))/(2.0*dx)+
     +           vs(i,j,k)*(c(i,j+1,k)-c(i,j-1,k))/(2.0*dy)+
     +           ws(i,j,k)*(c(i,j,k-1)-c(i,j,k+1))/(2.0*dz))
     +        + diffx*(c(i-1,j,k)-2.0*c(i,j,k)+c(i+1,j,k))/(dx*dx)
     +        + diffy*(c(i,j+1,k)-2.0*c(i,j,k)+c(i,j-1,k))/(dy*dy)
     +        + diffz*(c(i,j,k+1)-2.0*c(i,j,k)+c(i,j,k-1))/(dz*dz)
     +        + gtilde(i,j,k)*c(i,j,k)
   20 continue
      return
      end


      subroutine jacob(mx,my,mz,nxh,nx,ny,nz,t,c,tm,tu,tl,ipar,ws,
     +                 gtilde)
c-------------------------------------------------------------------
c   the subroutine jacob calculates the tridiagonal jacobian
c   matrices needed by the odd-even hopscotch method.
c   these matrices are returned in the arrays tm (main diagonals),
c   tl (lower diagonals), and tu (upper diagonals).
c   the parameter ipar determines whether we are dealing with the
c   "odd" or the "even" points
c-------------------------------------------------------------------
      dimension c(0:mx,0:my,0:mz),ws(nx,ny,nz),gtilde(nx,ny,nz),
     +          tm(nxh,ny,nz),tu(nxh,ny,nz),tl(nxh,ny,nz)
      real lx,ly,lz
      common/domain/lx,ly,lz
      common/param/diffx,diffy,diffz
      common/grid/dx,dy,dz
      common/time/tp,tb
      common/coef2/gamma,pi
c------------------------------------------------------------
c   calculate the time dependent part of the velocity field
c   in vertical direction
c------------------------------------------------------------
      d=cos(2.0*pi*t/tp)
c------------------------------------------------------------
c   first, the jacobian elements corresponding to the water
c   surface are calculated
c------------------------------------------------------------
      do 10 j=1,ny
      ib=mod(j+ipar,2)+1
      lc=0
      do 10 i=ib,nx,2
         lc=lc+1
         tm(lc,j,1)=-2.0*(diffz/(dz*dz)+diffx/(dx*dx)+diffy/(dy*dy))+
     +              gtilde(i,j,1)+(2.0*diffz/dz-d*ws(i,j,1))/lz
         tu(lc,j,2)=2.0*diffz/(dz*dz)
   10 continue
c------------------------------------------------------------
c   next, the jacobian elements corresponding to "internal"
c   vertical points are calculated
c------------------------------------------------------------
      do 30 j=1,ny
         do 20 k=2,nz-1
         ib=mod(j+ipar,2)+1
         lc=0
         do 20 i=ib,nx,2
            lc=lc+1
            tm(lc,j,k)=-2.0*(diffz/(dz*dz)+diffx/(dx*dx)+
     +                        diffy/(dy*dy))+gtilde(i,j,k)
            tl(lc,j,k)=diffz/(dz*dz)-d*ws(i,j,k)/(2.0*dz)
            tu(lc,j,k+1)=diffz/(dz*dz)+d*ws(i,j,k)/(2.0*dz)
   20    continue
   30 continue
c-------------------------------------------------------------
c   finally, the jacobian elements corresponding to the bottom
c   are calculated
c-------------------------------------------------------------
      do 40 j=1,ny
      ib=mod(j+ipar,2)+1
      lc=0
      do 40 i=ib,nx,2
         lc=lc+1
         tm(lc,j,nz)=-2.0*(diffz/(dz*dz)+diffx/(dx*dx)+diffy/(dy*dy))+
     +               gtilde(i,j,nz)-(2.0*diffz/dz+d*ws(i,j,nz))/lz
         tl(lc,j,nz)=2.0*diffz/(dz*dz)
   40 continue
      return
      end


      subroutine sol(mx,my,mz,nx,ny,nz,t,c)
c---------------------------------------------------
c   the subroutine sol calculates the exact solution
c   according to formula (5.1b)
c---------------------------------------------------
      dimension c(0:mx,0:my,0:mz)
      real lx,ly,lz
      common/grid/dx,dy,dz
      common/domain/lx,ly,lz
      common/time/tp,tb
      common/coef2/gamma,pi
c
      rt=(2.0+cos(2.0*pi*t/tp))/4.0
      st=(2.0+sin(2.0*pi*t/tp))/4.0
      ft=4.0*t/(tb+t)
      do 10 j=1,ny
         ys=(j-1)*dy/ly
         py2=(ys-st)**2
         do 10 k=1,nz
            zs=-(k-1)*dz/lz
            do 10 i=1,nx
               xs=(i-1)*dx/lx
               px2=(xs-rt)**2
               c(i,j,k)=exp(zs-ft-gamma*(px2+py2))
   10 continue
      return
      end


      subroutine uspace(nx,ny,nz,us)
c------------------------------------------------------
c   in the subroutine uspace, the time-independent part
c   of the velocity field in x-direction is calculated
c   and stored in the array us
c------------------------------------------------------
      dimension us(nx,ny,nz)
      real lx,ly,lz
      common/domain/lx,ly,lz
      common/coef1/cf1,cf2,beta
      common/grid/dx,dy,dz
```

```
c
      do 10 j=1,ny
         ys=(j-1)*dy/ly
         do 10 k=1,nz
            zs=-beta*(k-1)*dz/lz
            sz=sin(zs)
            do 10 i=1,nx
               xs=(i-1)*dx/lx
               us(i,j,k)=cf1*sin(xs+ys)*sz
   10 continue
      return
      end


      subroutine vspace(nx,ny,nz,vs)
c--------------------------------------------------------------
c   in the subroutine vspace, the time-independent part
c   of the velocity field in y-direction is calculated
c   and stored in the array vs
c--------------------------------------------------------------
      dimension vs(nx,ny,nz)
      real lx,ly,lz
      common/domain/lx,ly,lz
      common/coef1/cf1,cf2,beta
      common/grid/dx,dy,dz
c
      do 10 j=1,ny
         ys=(j-1)*dy/ly
         do 10 k=1,nz
            zs=-beta*(k-1)*dz/lz
            sz=sin(zs)
            do 10 i=1,nx
               xs=(i-1)*dx/lx
               vs(i,j,k)=cf2*cos(xs+ys)*sz
   10 continue
      return
      end


      subroutine wspace(nx,ny,nz,ws)
c--------------------------------------------------------------
c   in the subroutine wspace, the time-independent part
c   of the velocity field in z-direction is calculated
c   and stored in the array ws
c--------------------------------------------------------------
      dimension ws(nx,ny,nz)
      real lx,ly,lz
      common/domain/lx,ly,lz
      common/coef1/cf1,cf2,beta
      common/grid/dx,dy,dz
c
      do 10 j=1,ny
         ys=(j-1)*dy/ly
         do 10 k=1,nz
            zs=-beta*(k-1)*dz/lz
            cz=lz*cos(zs)/beta
            do 10 i=1,nx
               xs=(i-1)*dx/lx
               ws(i,j,k)=cz*(cf1/lx*cos(xs+ys)-cf2/ly*sin(xs+ys))
   10 continue
      return
      end
      subroutine g(nx,ny,nz,t,gtilde,us,vs,ws,ipar)
c-------------------------------------------------------------
c   the subroutine g calculates the concentration-independent
c   part of the source term; these values are stored in the
c   array gtilde
c-------------------------------------------------------------
      dimension gtilde(nx,ny,nz),
     +          us(nx,ny,nz),vs(nx,ny,nz),ws(nx,ny,nz)
      real lx,ly,lz
      common/domain/lx,ly,lz
      common/param/diffx,diffy,diffz
      common/grid/dx,dy,dz
      common/time/tp,tb
      common/coef2/gamma,pi
c
      ss=sin(2.0*pi*t/tp)
      cc=cos(2.0*pi*t/tp)
      rt=(2.0+cc)/4.0
      st=(2.0+ss)/4.0
      fdot=4.0*tb/(tb+t)**2
      rdot=-0.5*pi*ss/tp
      sdot=+0.5*pi*cc/tp
c
      do 10 j=1,ny
         ys=(j-1)*dy/ly
         py=ys-st
         py2=py*py
         ib=mod(j+ipar,2)+1
         do 10 k=1,nz
         do 10 i=ib,nx,2
            xs=(i-1)*dx/lx
            px=xs-rt
            px2=px*px
            gtilde(i,j,k)=
     +              - fdot+2.0*gamma*(px*rdot+py*sdot)
     +              - cc*(us(i,j,k)*2.0*gamma*px/lx +
     +                    vs(i,j,k)*2.0*gamma*py/ly -
     +                    ws(i,j,k)/lz)
     +              - diffx*2.0*gamma*(2.0*gamma*px2-1.0)/(lx*lx)
     +              - diffy*2.0*gamma*(2.0*gamma*py2-1.0)/(ly*ly)
     +              - diffz/(lz*lz)
   10 continue
      return
      end


      subroutine dcdx(mx,my,mz,nx,ny,nz,t,c)
c-------------------------------------------------------------
c   using the Neumann boundary conditions in the x-direction,
c   the subroutine dcdx calculates the concentrations in the
c   auxiliary points, that are situated on a distance delta_x
c   from the physical boundary
c-------------------------------------------------------------
      dimension c(0:mx,0:my,0:mz)
      real lx,ly,lz
      common/domain/lx,ly,lz
      common/grid/dx,dy,dz
      common/coef2/gamma,pi
      common/time/tp,tb
c
      xsw=0.0
      xse=1.0
      rt=(2.0+cos(2.0*pi*t/tp))/4.0
c
      do 10 k=1,nz
      do 10 j=1,ny
         c(0,j,k)=c(2,j,k)-
     +             2.0*dx*c(1,j,k)*(-2.0*gamma*(xsw-rt)/lx)
         c(nx+1,j,k)=c(nx-1,j,k)+
     +                 2.0*dx*c(nx,j,k)*(-2.0*gamma*(xse-rt)/lx)
   10 continue
      return
      end


      subroutine dcdy(mx,my,mz,nx,ny,nz,t,c)
c-------------------------------------------------------------
c   using the Neumann boundary conditions in the y-direction,
c   the subroutine dcdy calculates the concentrations in the
c   auxiliary points, that are situated on a distance delta_y
c   from the physical boundary
c-------------------------------------------------------------
      dimension c(0:mx,0:my,0:mz)
      real lx,ly,lz
      common/domain/lx,ly,lz
      common/grid/dx,dy,dz
      common/coef2/gamma,pi
      common/time/tp,tb
c
      ysf=0.0
      ysb=1.0
      st=(2.0+sin(2.0*pi*t/tp))/4.0
c
      do 10 k=1,nz
      do 10 i=1,nx
         c(i,0,k)=c(i,2,k)-
     +             2.0*dy*c(i,1,k)*(-2.0*gamma*(ysf-st)/ly)
         c(i,ny+1,k)=c(i,ny-1,k)+
     +                 2.0*dy*c(i,ny,k)*(-2.0*gamma*(ysb-st)/ly)
   10 continue
      return
      end


      subroutine dcdz(mx,my,mz,nx,ny,nz,t,c)
c-------------------------------------------------------------
c   using the Neumann boundary conditions in the z-direction,
c   the subroutine dcdz calculates the concentrations in the
c   auxiliary points, that are situated on a distance delta_z
c   from the physical boundary
c-------------------------------------------------------------
      dimension c(0:mx,0:my,0:mz)
      real lx,ly,lz
      common/domain/lx,ly,lz
      common/grid/dx,dy,dz
c
      do 10 j=1,ny
      do 10 i=1,nx
         c(i,j,0)=c(i,j,2)+2.0*dz*c(i,j,1)/lz
         c(i,j,nz+1)=c(i,j,nz-1)-2.0*dz*c(i,j,nz)/lz
   10 continue
      return
      end
      subroutine TRI3S5 (NPP, al3, ad3, au3, bb3, ind, n1, n2, n3,
     +                   Alpha, NewJac, NewAlp, ifail)
c-----------------------------------------------------------------------
c   TRI3S5   Subroutine makes LD^{-1}U-decompositions of tridiagonal
c            matrices T and solves Tx=b.
c            T is formed by T=I-alpha*JAC, from a given real value alpha and
c            matrix JAC.
c            The computations are distributed over the NPP available
c            parallel processors.
c            Logical parameters NewJac and NewAlp serve the possibility to
c            input previously computed decompositions.
c            The value of ind (1, 2 or 3) corresponds to the coupling of
c            unknowns: either in x-, y- or z-direction.
c-----------------------------------------------------------------------
c   Parameters:
c     Input: matrices in al3(), ad3(), au3(), right-hand sides in bb3()
c     Output: solutions in bb3()
c-----------------------------------------------------------------------
      real al3(n1, n2, n3), ad3(n1, n2, n3),
     +     au3(n1, n2, n3), bb3(n1, n2, n3)
      real Alpha
      logical NewJac, NewAlp
      integer NPP, ind, n1, n2, n3, ifail
c---------------
c   Declarations
c---------------
      integer n12,n23,ix,iz,first,last,ipp,ntasks
      real zero
      external TRI3P1, TRI3P3
c
      data zero/ 0.0 /
      ifail = 0
c-------------------------------
c   Test for error in parameters
c-------------------------------
      if ((ind .lt. 1) .or. (ind .gt. 3)) then
         ifail = 5
         return
      end if
      n12  = n1*n2
      n23  = n2*n3
      if (ind .eq. 1) then
CFPP$ CNCALL L
         do 10, ipp = 1, NPP
            first = (n23*(ipp-1)+NPP-1)/NPP+1
            ntasks= ((n23*ipp-1)/NPP+1) - first + 1
            last  = first + ntasks - 1
            call TRI3P1 (al3(1, first, 1), ad3(1, first, 1),
     +         au3(1, first, 1), bb3(1, first, 1), n1, ntasks,
     +         Alpha, NewJac, NewAlp, ifail)
  10        continue
      else if (ind .eq. 2) then
CFPP$ NODEPCHK
CFPP$ CNCALL L
         do 30, ipp = 1, NPP
            first = (n3*(ipp-1)+NPP-1)/NPP+1
            ntasks= ((n3*ipp-1)/NPP+1) - first + 1
            last  = first + ntasks - 1
c-------------------------------------------
c   Set  al3(ix,1,iz) and au3(ix,1,iz) to zero
c-------------------------------------------
            if (NewJac) then
```

```
               do 20, iz = first+1, last
                  do 20, ix = 1, n1
                     al3(ix,1,iz) = zero
   20                au3(ix,1,iz) = zero
            end if
            call TRI3P3 (al3(1, 1, first), ad3(1, 1, first),
     +         au3(1, 1, first), bb3(1, 1, first), n1, n1, n2*ntasks,
     +         Alpha, NewJac, NewAlp, ifail)
   30       continue
      else
CFPP$ CNCALL L
         do 40, ipp = 1, NPP
            first = (n12*(ipp-1)+NPP-1)/NPP+1
            ntasks= ((n12*ipp-1)/NPP+1) - first + 1
            last  = first + ntasks - 1
            call TRI3P3 (al3(first, 1, 1), ad3(first, 1, 1),
     +         au3(first, 1, 1), bb3(first, 1, 1), ntasks, n12, n3,
     +         Alpha, NewJac, NewAlp, ifail)
   40       continue
      end if
      return
      end


      subroutine TRI3P1 (al3, ad3, au3, bb3, n1, n23,
     +                   Alpha, NewJac, NewAlp, ifail)
c-----------------------------------------------------------------------
c   TRI3P1   Auxiliary subroutine to TRI3S5 for case ind=1.
c   Parameters:
c     Input: matrices in al3(), ad3(), au3(), right-hand sides in bb3()
c     Output: solutions in bb3()
c-----------------------------------------------------------------------
      integer n1, n23, ifail
      real al3(n1, n23), ad3(n1, n23), au3(n1, n23), bb3(n1, n23)
      real Alpha
      logical NewJac, NewAlp
c---------------
c   Declarations
c---------------
      integer i, ix
      real tmp, factor, one
      data one / 1.0 /
c
      ifail = 0
c-----------------------------------------------------------------------
c   Input: matrices in al3(2:n1,iy,iz), ad3(1:n1,iy,iz), au3(2:n1,iy,iz),
c          right-hand side in bb3(1:n1,iy,iz).
c          al3(1,iy,iz)=0.0 and au3(1,iy,iz)=0.0 not required.
c-----------------------------------------------------------------------
      if (NewJac.or.NewAlp) then
c---------------------------------
c   Make T=I-alpha*J and decompose
c---------------------------------
         if (.not.NewJac) then
c-----------------------------------------------------
c   Form I-oldalpha*J from L*D^{-1}*U, and form -oldalpha*J
c-----------------------------------------------------
            do 10, ix = n1, 2, -1
               do 10, i = 1, n23
                  ad3(ix,i) = one/ad3(ix,i) + al3(ix,i)*au3(ix,i) - one
                  al3(ix,i) = al3(ix,i) / ad3(ix-1,i)
   10          continue
```

```
            do 20, i = 1, n23
               ad3(1,i) =  one / ad3(1,i) - one
   20          continue
            factor = Alpha/al3(1,1)
         else
c----------------------
c   Form only I-alpha*J
c----------------------
            factor = -Alpha
         end if
         al3(1,1) = Alpha
c-----------------------------------------------------------------
c   Form I+factor*matrix, decomposition and forward substitution
c-----------------------------------------------------------------
         do 30, i = 1, n23
            ad3(1,i) = one / (ad3(1,i) * factor + one)
   30       continue
         do 40, ix = 2, n1
            do 40, i = 1, n23
               tmp      = al3(ix,i) * ad3(ix-1,i) * factor
               au3(ix,i) = au3(ix,i) * factor
               al3(ix,i) = tmp
               ad3(ix,i) = one / (ad3(ix,i) * factor + one
     +                        - tmp * au3(ix,i))
               bb3(ix,i) = bb3(ix,i) - tmp * bb3(ix-1,i)
   40       continue
      else
c----------------------------
c   Only forward substitution
c----------------------------
         do 50, ix = 2, n1
            do 50, i = 1, n23
               bb3(ix,i) = bb3(ix,i) - al3(ix,i)*bb3(ix-1,i)
   50    continue
      end if
c---------------------------------
c   Proceed with back substitution
c---------------------------------
      do 60, i = 1, n23
         bb3(n1,i) = bb3(n1,i) * ad3(n1,i)
   60    continue
      do 70, ix = n1-1, 1, -1
         do 70, i = 1, n23
            bb3(ix,i) = (bb3(ix,i) - au3(ix+1,i)*bb3(ix+1,i))*ad3(ix,i)
   70    continue
      return
      end


      subroutine TRI3P3 (al3, ad3, au3, bb3, ntsks, n12, n3,
     +                   Alpha, NewJac, NewAlp, ifail)
c--------------------------------------------------------------------
c   TRI3P3   Auxiliary subroutine to TRI3S5 for case ind=2 and ind=3.
c   Parameters:
c     Input: matrices in al3(), ad3(), au3(), right-hand sides in bb3()
c     Output: solutions in bb3()
c--------------------------------------------------------------------
      integer ntsks, n12, n3, ifail
      real al3(n12,n3), ad3(n12,n3), au3(n12,n3), bb3(n12,n3)
      real Alpha
      logical NewJac, NewAlp
```

```
c---------------
c   Declarations
c---------------
      integer i,iz
      real tmp, factor, one
      data one / 1.0 /
c
      ifail = 0
c------------------------------------------------------------------------
c   Input: matrix in al3(i,2:n3), ad3(i,1:n3), au3(i,2:n3),
c          right-hand side in bb3(i,1:n3).
c          al3(i,1)=0.0 and au3(i,1)=0.0 must be set in TRI3S5 for ind=2.
c------------------------------------------------------------------------
      if (NewJac.or.NewAlp) then
c--------------------------------
c   Make T=I-alpha*J and decompose
c--------------------------------
         if (.not.NewJac) then
c-----------------------------------------------------
c   Form I-oldalpha*J from L*D^{-1}*U, and form -oldalpha*J
c-----------------------------------------------------
            do 10, iz = n3, 2, -1
               do 10, i = 1, ntsks
                  ad3(i,iz) = al3(i,iz)*au3(i,iz) + one/ad3(i,iz) - one
                  al3(i,iz) = al3(i,iz) / ad3(i,iz-1)
   10          continue
            do 20, i = 1, ntsks
               ad3(i,1) = one / ad3(i,1) - one
   20          continue
            factor = Alpha/al3(1,1)
         else
c----------------------
c   Form only I-alpha*J
c----------------------
            factor = -Alpha
         end if
         al3(1,1) = Alpha
c------------------------------------------------------------
c   Form I+factor*matrix, decomposition and forward substitution
c------------------------------------------------------------
         do 30, i = 1, ntsks
            ad3(i,1) =  one / (ad3(i,1) * factor + one)
   30       continue
         do 40, iz = 2, n3
            do 40, i = 1, ntsks
               tmp       = al3(i,iz) * ad3(i,iz-1) * factor
               au3(i,iz) = au3(i,iz) * factor
               al3(i,iz) = tmp
               ad3(i,iz) = one / (ad3(i,iz) * factor + one
     +                     - tmp * au3(i,iz))
               bb3(i,iz) = bb3(i,iz) - tmp * bb3(i,iz-1)
   40       continue
      else
c-------------------------
c   Only forward substitution
c-------------------------
         do 50, iz = 2, n3
            do 50, i = 1, ntsks
               bb3(i,iz) = bb3(i,iz) - al3(i,iz) * bb3(i,iz-1)
   50    continue
      end if
```

```
c----------------------------------
c   Proceed with back substitution
c----------------------------------
      do 60, i = 1, ntsks
         bb3(i,n3) = bb3(i,n3) * ad3(i,n3)
   60    continue
      do 70, iz = n3-1, 1, -1
         do 70, i = 1, ntsks
            bb3(i,iz) = (bb3(i,iz) - au3(i,iz+1)*bb3(i,iz+1))*ad3(i,iz)
   70    continue
      return
      end
```