Research and Consulting Services in the Environmental and Earth Sciences

ENVAIR 96-001
July 1996

Pluvius II User's Manual

Principal Authors
J. M. Hales
R. C. Easter*

Prepared for the U.S. Department of Energy
under Grant No. DE-FG06-93ER61552
and Contract No. 960182402

by
Envair
4811 W. 18th Avenue
Kennewick, Washington 99337

*Pacific Northwest National Laboratory

DISCLAIMER

ENVAIR
for the
UNITED STATES DEPARTMENT OF ENERGY
under Grant DE-FG06-93ER61552
and Contract 960182402

Copies of this report and the Pluvius II Fortran code are available on anonymous ftp, at odysseus.owt.com. Retrieve file readme.pluviusII for further instructions.

ENVAIR 96-001
July 1996

PLUVIUS II USER'S MANUAL

J. M. Hales
R. C. Easter*

Prepared for
the U.S. Department of Energy
under Grant No. DE-FG06-93ER61552
and Contract No. 960182402

Envair
Kennewick, Washington 99337

*Pacific Northwest National Laboratory

ABSTRACT

This user's manual has been prepared to guide the reader in the setup and execution of Pluvius II, a Fortran computer code which has been prepared for general mechanistic analysis of a variety of air-pollution and atmospheric-chemistry problems. Pluvius II is a general-purpose code for modeling pollutants in the gas phase and in conjunction with cloud and precipitation systems. It is based on Eulerian representations of conservation equations for chemical species, energy, and the physical media (e.g., air, cloud water, rain water, ice . . .) in which the chemical species reside. Because energy and moisture conservation equations are included, the code is capable of simulating cloud and storm formation, and can deal directly with the attachment, wet-chemistry, and deposition processes associated with precipitating systems.

The code has been structured to allow considerable flexibility in its use. One-, two- or three-dimensional simulations can be performed, and selection of modeled chemical species, physical media, physicochemical interaction systems, spatial/temporal domain, and grid spacing is at the option of the user. This user's manual provides listings of the code for two illustrative examples, which are intended to guide the reader to a level of understanding that is sufficient for application to a variety of extended problems.

ACKNOWLEDGEMENTS

Pluvius II originates from Stockholm University, where one of us (Hales) spent a summer several years ago building the code's framework and writing the first version of its numerical integration scheme. The code has evolved significantly since that time, with both of us making needed modifications and corrections, applying it for a variety of precipitation-scavenging evaluations and, finally, writing this user's manual.

It was our original intent to place the code promptly in the hands of the extended user community, and we hadn't expected that this last part to take so long. Now that the user's manual is indeed complete, however, we would like to express our sincere thanks to the institutions and individuals that have made this work possible. First, we would like to thank Lennart Granat, Henning Rodhe, and other staff members of the Meteorological Institute at Stockholm University for their guidance and encouragement, and for providing the sanctuary necessary for early parts of this effort to be accomplished. In addition we express our sincere appreciation to our primary sponsors, the Office of Health and Environmental Research of the U.S. Department of Energy. The continuing guidance and expressions of support given us by David Slade, Ari Patrinos, Mike Riches, Michelle Broido, and Rickey Petty were essential ingredients of this outcome. We hope that the code's future application by the user community will be extensive and will reflect the confidence that we have received from these people.

Pluvius II is a product of the DOE Atmospheric Chemistry Program.

CONTENTS

Section **Page**

1 **Introduction** **1-1**

2 **Governing Equations** **2-1**
- Material Balances **2-1**
- Energy Balance **2-2**
- Generalized Form **2-5**
- Boundary Conditions **2-5**
- Initial Conditions **2-6**
- Dependent-Variable Classes **2-7**

3 **Numerical Approximation Methods** **3-1**
- Operator Splitting **3-1**
- Numerical Integration of Transport Terms **3-2**
- Numerical Integration of Physicochemical Transformation Terms **3-2**
- Numerical Filtering Techniques **3-2**
- Performance of Numerical-Integration Technique **3-3**

4 **Overview of Code Properties** **4-1**

5 **Example Computations** **5-1**
- Description of Case Examples **5-1**
- Executing the Case Example Codes **5-4**

6 **Detailed Code Description** **6-1**
- Introduction **6-1**
- Overview of Main Program and First-Level Subroutines **6-2**
- Overview of Second- and Lower-Level Subroutines **6-11**
  - *Initialization and Grid-Setup Operations* **6-11**
  - *Transport-Integration Operations* **6-12**
  - *Numerical Filtering Operations* **6-15**
  - *Transformation-Integration Operations* **6-15**
  - *Utility Routines for Calculation of Thermodynamic Properties* **6-17**
  - *General Utility Subroutines* **6-18**

LIST OF ILLUSTRATIONS

Figure **Page**

4.1 Example Interaction Diagrams for Code-Setup.............................4-2

5.1 Plot of Wind Field Used for Example Simulations.........................5-2

5.2 Selected Results of Case Example A Simulation at......................5-6 10 Hours Simulation Time

5.3 Selected Results of Case Example B Simulation at......................5-6 10 Hours Simulation Time

LIST OF TABLES

Table Page

5.1 Summary of Physicochemical Parameterizations..................... 5-3
Applied for Case-Example Simulations A and B

5.2 Boundary Conditions Applied for Case-Example..................... 5-4
Simulations A and B

6.1 Pluvius II Code Variable and Array Definitions........................ 6-19

6.2 Listing of Main Program pluvius.. 6-27

6.3a Listing of Include File pluvius.com for Case Example A.......... 6-29

6.3b Listing of Include File pluvius.com for Case Example B.......... 6-37

6.4a Listing of Subroutine inptgn for Case Example A.................... 6-44

6.4b Listing of Subroutine inptgn for Case Example B.................... 6-48

6.5 Listing of Subroutine femset.. 6-54

6.6a Listing of Subroutine init for Case Example A....................... 6-57

6.6b Listing of Subroutine init for Case Example B....................... 6-61

6.7 Listing of Subroutine wind.. 6-63

6.8 Listing of Subroutine printi.. 6-66

6.9 Listing of Subroutine print.. 6-67

6.10 Listing of Subroutine matbal.. 6-67

6.11 Listing of Subroutine diff... 6-71

6.12 Listing of Subroutine core 6-72

6.13a Listing of Subroutine cleanup for Case Example A.............. 6-74

6.13b Listing of Subroutine cleanup for Case Example B.............. 6-76

6.14 Listing of Subroutine collect.. 6-77

6.15 Listing of Subroutine restart.. 6-78

6.16 Listing of Subroutine cloudinit.. 6-80

6.17 Listing of Subroutine inflowinit... 6-82

6.18 Listing of Subroutine core00.. 6-83

6.19 Listing of Subroutine grdxyz.. 6-84

6.20 Listing of Subroutine sinteg.. 6-86

6.21 Listing of Subroutine propss.. 6-88

6.22 Listing of Subroutine coefss.. 6-90

6.23 Listing of Subroutine bounds.. 6-92

6.24 Listing of Subroutine loads.. 6-93

6.25 Listing of Subroutine transport... 6-95

6.26 Listing of Subroutine xinteg.. 6-98

6.27 Listing of Subroutine propsx.. 6-100

6.28 Listing of Subroutine coefsx.. 6-101

6.29 Listing of Subroutine boundx.. 6-104

6.30 Listing of Subroutine loadx.. 6-105

6.31 Listing of Subroutine yinteg.. 6-107

6.32 Listing of Subroutine propsy... 6-108

6.33 Listing of Subroutine coefsy... 6-110

6.34 Listing of Subroutine boundy... 6-113

6.35 Listing of Subroutine loady... 6-114

6.36 Listing of Subroutine sfilter... 6-116

6.37 Listing of Subroutine xfilter... 6-117

6.38 Listing of Subroutine yfilter... 6-119

6.39 Listing of Subroutine bndfilt... 6-120

6.40 Listing of Subroutine foresfilt... 6-121

6.41 Listing of Subroutine chmint... 6-124

6.42 Listing of Subroutine odeint... 6-125

6.43a Listing of Subroutine gen for Case Example A.................... 6-134

6.43b Listing of Subroutine gen for Case Example B.................... 6-138

6.44 Listing of Function aircon... 6-148

6.45 Listing of Function esat... 6-148

6.46 Listing of Function esati... 6-149

6.47 Listing of Function sat... 6-149

6.48 Listing of Function ssat... 6-150

6.49 Listing of Function rlapse... 6-150

6.50 Listing of Subroutine henry... 6-151

6.51 Listing of Subroutine tfrmth... 6-152

6.52 Listing of Subroutine integrate... 6-157

6.53 Listing of Subroutine tridag.. 6-158

6.54 Listing of Subroutine sbstep.. 6-159

6.55 Listing of Subroutine ijpack.. 6-159

6.56 Listing of Subroutine ijunpack.. 6-161

6.57 Subroutine Summary and Location Chart............................. 6-162

Section 1

Introduction

The objective of this user's manual is to guide the reader in the setup and execution of Pluvius II, a Fortran computer code which has been prepared for general mechanistic analysis of a variety of air-pollution and atmospheric-chemistry problems. In essence, the code describes multiphase pollution behavior in a volume of the atmosphere, which is specified by the user in terms of a Cartesian (x,y) horizontal grid system and an arbitrary vertical coordinate which may be Cartesian (z) in form or any of a variety of alternatives.[1] This description is formulated around conservation equations for air, water, and an arbitrary number of trace gases and/or aerosols, combined with an additional conservation equation for energy.

Pluvius II differs from conventional "airshed" chemical models in one important respect: because it incorporates balance equations for energy as well as a variety of water classes, it is able to simulate cloud and precipitation formation in a natural manner. This in turn provides a convenient computational vehicle to simulate a variety of gas-liquid pollutant-exchange processes, in addition to aqueous-phase reaction chemistry and precipitation scavenging. As noted above, Pluvius II is designed to simulate three-dimensional systems; on the other hand it can be used in one- and two-dimensional simulations as well, if desired by the user.

A number of codes of this general class have been described in the open literature during the last several years (e.g. Hegg et al. 1984, Molencamp 1983, Hales 1981, 1982, Easter and Hales 1982, 1984, Tremblay and Leighton 1986). To our knowledge, however, Pluvius II is the only multidimensional code of this type that has been designed with the intent of user's-manual documentation for extended use by the atmospheric research community. In this context we note that Pluvius II's one-dimensional predecessors, Pluvius versions 3.1 and 5, have been described in user's-manual form and circulated rather extensively. Pluvius II differs from these predecessors in the obvious sense that it allows simulations in up to three dimensions. It also contains much more robust numerical algorithms and generally provides a more convenient computational platform. On the other hand, it is quite similar to its predecessors in the sense that its modular form provides a flexible computational platform where component subroutines such as kinetics schemes, cloud-physics descriptions, wind-fields, and so-forth can be replaced, compiled, and executed in a straightforward and convenient manner.

---

[1]**For a discussion of alternate vertical coordinate systems, see Haltiner and Williams (1980).**

The governing equations for Pluvius II have been described previously in a journal article (Hales 1989). Furthermore, the case examples used in this manual are based on those in that article, and we strongly recommend that the reader consult this reference frequently as he or she proceeds through this manual.

The following five sections of this user's manual are intended to provide the reader with a knowledge of the theoretical foundations of Pluvius-II and the capability to operate the model, as well as to document and explain the model's code in sufficient detail to facilitate extended future modification to meet specific user needs. Section 2 describes the Pluvius II's basic governing equations and initial and boundary conditions, and Section 3 provides a brief overview of the numerical-integration methods employed by the code to obtain the corresponding solutions.

Section 4 presents a basic overview of code properties. Intentionally brief, this overview is designed to provide the user with a "jump-start" to Section 5, which describes an application of the code to simulate chemistry and wet removal within a two-dimensional storm characterization. Detailed description of the Fortran code appears in Section 6, and is intended to supply the in-depth information required for progressively ambitious extensions of the code by a widespread user community.

Section 2

Governing Equations

2.1 Material Balances

As indicated in Section 1, the Pluvius II is able to accommodate as many chemical species as desired by the user, as computer resources permit. Chemical species are partitioned among a group of physical “media,” in which the species may reside. The number and classification of media are determined by the user; for example, a particular simulation may classify the gas phase as one medium and condensed atmospheric water as a second. A more elaborate simulation may partition the condensed water into the separate “media” of cloud-water, ice, and rain. Still more complex simulations may subdivide water groups into additional classifications, based on hydrometeor size and/or morphology. Regardless of the choice of media, all species-medium pairs are treated as dependent variables in individual material balances of the simulation. Derivation of these balances is based on the form (cf. Bird, Stewart, and Lightfoot 1960)

$$\frac{\partial c_{A,m}}{\partial t} = - \quad (c_{A,m}\mathbf{v}_{A,m}) + R_{A,m}, \tag{2.1}$$

where $c_{A,m}$ is the concentration of chemical species A in medium m (gram-moles of species A bound in medium per cm^3 of total space), and $\mathbf{v}_{A,m}$ is its velocity vector in Cartesian coordinates. $R_{A,m}$ is a source-sink term which accounts for the contributions of physicochemical processes to the overall balance.

Combining equation (2.1) with the continuity equation for air, time-smoothing, applying gradient-transport theory in the conventional manner, and expressing the dependent variables as mixing ratios rather than concentrations, yields the form

$$\frac{Dr_{A,m}}{Dt} = -\frac{1}{c}\frac{\partial}{\partial z}(c\, r_{A,m} w_m) + \frac{1}{c} \quad (c\, \mathbf{K}_m \quad r_{A,m}) + \frac{R_{A,m}}{c}, \tag{2.2}$$

where

$\frac{D}{Dt}$ is the conventional substantial-derivative operator, $\frac{\partial}{\partial t} + \mathbf{v} \quad$;

$r_{A,m} = c_{A,m}/c$ is the time-smoothed pollutant mixing ratio (molar basis);

$\mathbf{v}$ is the time-smoothed wind-velocity vector;

c is the molar concentration of air;

w_m is the vertical fall velocity of m-bound pollutant in stagnant air; and

$\mathbf{K}_m$ is the associated turbulent diffusivity tensor, which is assumed to be diagonal.

2.2 Energy Balance

The energy balance employed by the code is based on the form (cf. Bird, Stewart and Lightfoot 1960)

$$c\,C_v \frac{DT}{Dt} = -p \quad \mathbf{v} + \Gamma\,, \tag{2.3}$$

where

C_v is the specific heat of air at constant volume (molar basis),

T is the absolute temperature,

p is the local pressure, and

Γ accounts for energy input arising from latent-heating effects.

As shown by Hales (1989) this may be combined with the hydrostatic and ideal-gas equations, and time-smoothed to provide the form

$$\frac{DT}{Dt} = w\,\gamma_d + \frac{1}{c} \quad \left[c\,\mathbf{K}_H\ (\quad T - \mathbf{k}\,\gamma_d\right] + \frac{1}{c\,C_p}\left[\lambda_c r_c^* + (\lambda_c + \lambda_f) r_d^* + \lambda_f r_f^*\right] \tag{2.4}$$

where

w is the z-component of the wind-velocity vector;

$\mathbf{K}_H$ represents the diffusivity tensor for sensible heat;

$\mathbf{k}$ is the vertical unit vector;

λ_c and λ_f are the latent heat of water-vapor condensation to liquid and latent heat of fusion of liquid water to ice, respectively;

r_c^*, r_d^*, and r_f^* are the respective molar rates of phase transformation by water-vapor condensation to liquid, water-vapor deposition to ice, and freezing of liquid water;

C_p is the specific heat of air at constant pressure (molar basis); and

γ_d is the dry adiabatic lapse rate for air, given by

$$\gamma_d = -\frac{g\, M_{air}}{g_c C_p}, \tag{2.5}$$

where g represents the acceleration by gravity, g_c is the international gravitational constant,[1] and M_{air} is the molecular weight of air.

In regions of the atmosphere where cloud liquid water exists, it is convenient to combine equation (2.4) with a material balance for water of the form (2.2), with the added assumption that water-vapor supersaturation is approximately zero. This corresponds to simultaneous heat and mass transfer under “zero-supersaturation” conditions, and may be represented by the form

$$\chi_1 \frac{DT}{Dt} = \frac{\lambda_c^2}{c\, R\, C_p} \quad \left(\frac{c\, r_{vs}}{T^2} \mathbf{K}_v \quad T\right) + w\, \gamma_d \left(1 + \frac{r_{vs}\, \lambda_c}{RT}\right)$$
$$- \frac{\lambda_c}{c} \frac{\partial}{\partial z} \left(\frac{c\, r_{vs} \gamma_d K_{v,z}}{RT}\right) + \frac{\lambda_f}{c\, C_p} (r_d^* + r_f^*) + \frac{1}{c} \quad \left[c\, \mathbf{K}_H \ \left(\ T - \mathbf{k}\, \gamma_d\right)\right] . \tag{2.6}$$

Here

$\mathbf{K}_V$ represents the diffusivity tensor for water vapor,

r_{vs} is the saturation mixing ratio of water vapor,

[1]g_c in this equation has a value of unity, and units of gm-cm/dyne-sec^2. Although this practice is by no means universal, many scientific and engineering fields insert g_c to satisfy the conversion from mass to force units in Newton's second law, which appears implicitly in Equation (2.5). We will follow this practice throughout this user's manual.

R is the gas-law constant (molar basis), and

$$\chi_1 = 1 + \frac{r_v \lambda_c^2}{C_p R\, T^2}\ .$$

Owing to its relative ease of use, equation (2.6) is preferable for reactive scavenging calculations, unless cloud-supersaturation phenomena are expected to play an uncommonly significant role in the scavenging process, or if the model is being employed to assess weather-modification effects. One should note with caution, however, that this equation applies only for conditions where local cloud water exists: computations in regions where no cloud water exists should apply equation (2.4) directly.

Because of numerical-computation difficulties associated with discontinuities at cloud boundaries, it is sometimes convenient to convert the energy balance to a form that allows expression in terms of a dependent variable that has a natural tendency toward smoothness regardless of cloud position. This may be accomplished using a method similar to that used by Tripoli and Cotton (1981) by defining a "cloud-equivalent potential temperature," θ_c, such that

$$\theta_c = \theta_d \left(1 - \frac{\lambda_c r_c}{C_p T}\right) \tag{2.7}$$

where θ_d is the conventional potential temperature, which may be expressed by the form

$$\theta_d = T\left(\frac{1000\ \text{mb}}{p}\right)^{R/C_p} \tag{2.8}$$

and r_c is the mixing ratio of cloud water. θ_c is a conserved entity, which represents energy-balance effects of expansion work and the latent heat of condensation to cloud droplets. As a consequence, the energy balance can be expressed in terms of θ_c as a dependent variable by the form

$$\frac{D\theta_c}{Dt} = \frac{1}{c}\quad (c\, K_{\theta_c} \qquad \theta_c) + \frac{1}{c\, C_p}\left[\lambda_c r_c^{**} + (\lambda_c + \lambda_f) r_d^{*} + \lambda_f r_f^{*}\right] \tag{2.9}$$

where r_c** accounts for water condensation/evaporation to/from raindrops, but not cloud water.[2]

[2]**See Hales (1989) for details.**

2.3 Generalized Form

The governing equations (2.2), (2.4), (2.6) and (2.9) can be expressed in the general form

$$\frac{\partial \zeta}{\partial t} = \frac{\partial}{\partial x}(A_x \frac{\partial \zeta}{\partial x}) + B_x \frac{\partial \zeta}{\partial x} + \frac{\partial}{\partial y}(A_y \frac{\partial \zeta}{\partial y}) + B_y \frac{\partial \zeta}{\partial y}$$
$$+ \frac{\partial}{\partial z}(A_z \frac{\partial \zeta}{\partial z}) + B_z \frac{\partial \zeta}{\partial z} + C_z \zeta + G \qquad (2.10)$$

where the dependent variable ζ denotes either $r_{A,m}$, T, or θ_C, and the coefficients $A_i(x,y,z,t)$, $B_i(x,y,z,t)$, $C_i(x,y,z,t)$ and G(x,y,z,t) can be determined from the pollutant- and energy-balance equations in an obvious and straightforward manner. Equation (2.10) is the basic form whose solution is approximated by the code's numerical approximation scheme.

2.4 Boundary Conditions

Boundary conditions must be specified at each end of the computational domain for every active dimension of the code; thus in a three-dimensional simulation, conditions are imposed at six boundaries. Flux boundary conditions are used at each boundary, and take the general form of

$$v_i \, c \, r_{A,m} - c \, K_{m,i} \frac{\partial r_{A,m}}{\partial x_i} = \text{prescribed flux of species A in medium m in direction i, (moles cm}^{-2}\text{ sec}^{-1}) \qquad (2.11)$$

$$v_i \, c \, C_p \, \theta_c - c \, C_p K_{H,i} \frac{\partial \theta_c}{\partial x_i} = \text{prescribed energy flux in direction i, (erg cm}^{-2}\text{ s}^{-1}) \qquad (2.12)$$

where the energy flux in (2.12) is essentially the flux of moist static energy.

The x, y, and upper z boundaries are treated as open boundaries, across which air is generally free to move in either direction. Here the fluxes are not prescribed directly using equations (2.11) and (2.22). At points along a boundary where the velocity component is either zero or into the computational domain, the inflow fluxes are stipulated on the basis of prescribed boundary mixing ratios and temperatures ($r_{A,m,infl}$ and $\theta_{A,m,infl}$) and velocities:

prescribed inflow flux of $r_{A,m}$ in direction i = v_i c $r_{A,m,infl}$ (2.13)

prescribed inflow energy flux in direction i = v_i c $C_p\,\theta_{c,infl}$ (2.14)

At points along a boundary where the flow is out of the computational domain, the fluxes are computed automatically using the values of $r_{A,m}$ and θ_c at model grid points adjacent to the boundary.

The lower z boundary is generally at the earth's surface and thus is not "open" in the sense described above. For gas and aerosol species, the prescribed surface flux can include an emission term as well as a dry-deposition term, provided by some parameterization that involves surface and boundary-layer properties and the value of $r_{A,m}$ adjacent to the surface. Similarly, the surface energy flux might be provided by some parameterization for surface energy transfer.

For species associated with precipitation (e.g., rainwater and sulfate ion in rain) which have appreciable fall velocities, the lower z-boundary is effectively an open boundary, where outflow fluxes are computed automatically in the manner described above.

2.5 Initial Conditions

Initial conditions, in the form of temperatures and mixing ratios, must be supplied at each point of the computational domain to initialize the simulation; that is,

$$r_{A,m}(x,y,z,t_0) = r_{A,m,0}(x,y,z) \text{ at } t = t_0 \qquad (2.13)$$

$$T(x,y,z,t_0) = T_0(x,y,z) \quad \text{at } t = t_0 \qquad (2.14)$$

where the subscript 0 pertains to the model’s start time.

2.6 Dependent-Variable Classes

The dependent variables r and T (or θ_c) can be assigned to two basic computational classes, depending on whether they: (a) are to be computed directly from equations (2.2), (2.4), (2.6), or (2.9) [general equation (2.10)], or (b) are to be supplied by other means. The first computational class shall be referred to as transported variables, since these entities are transported in space in conformance with the model's governing equations. The second class includes both steady-state variables (which are calculated algebraically within the code) and those variables that are supplied by external “drivers” (e.g., prescribed from observations). Because these variables can be considered to be constrained, either to the behavior of the transported variables or by external factors, they shall be referred to as “constrained variables” in the present discussion. The code provides for the formal declaration of both variable types.

Section 3

Numerical Approximation Methods

3.1 Operator Splitting

An important code feature is its use of “split” operators. Using this technique the numerical approximations at any locality (x,y,z) are considered to be mechanistically and dimensionally independent over short intervals of time; thus the total change of the dependent variable $\partial\zeta/\partial t$ is assumed to be derivable from the local independent operators for transport in the x, y, and z directions, and for physicochemical transformation:

$$\left.\frac{\partial\zeta}{\partial t}\right|_x, \quad \left.\frac{\partial\zeta}{\partial t}\right|_y, \quad \left.\frac{\partial\zeta}{\partial t}\right|_z, \text{ and } \left.\frac{\partial\zeta}{\partial t}\right|_{\text{transformation}} \tag{3.1}$$

Actual implementation of the split-operator system within the code takes the form

$$\begin{aligned}\Delta\zeta\Big|_{\substack{\text{total,}\\ 2\Delta t}} = {} & \Delta\zeta\Big|_{\substack{x,\\ \Delta t}} + \Delta\zeta\Big|_{\substack{y,\\ \Delta t}} + \Delta\zeta\Big|_{\substack{z,\\ \Delta t}} \\ & + \Delta\zeta\Big|_{\substack{\text{transformation,}\\ 2\Delta t}} + \Delta\zeta\Big|_{\substack{z,\\ \Delta t}} + \Delta\zeta\Big|_{\substack{y,\\ \Delta t}} + \Delta\zeta\Big|_{\substack{x,\\ \Delta t}}\end{aligned} \tag{3.2}$$

where the individual $\Delta\zeta$ terms are interpreted as the local changes over the indicated intervals of time. This operator-splitting procedure, which is also known as the “locally one-dimensional” approximation, was first introduced by Yanenko (1971), and has gained rather general acceptance (cf. Lapidus and Pinder, 1982). Air-pollution codes applying this approximation in the past have included the Urban Airshed Model (UAM) (Reynolds, Seinfeld and Roth, 1973) and the Sulfur Transport Eulerian Model (STEM) Charmichael and Peters, 1984).

In practice the code deals with the energy balance (equation 2.9) by computing the advection and diffusion of θ_c in the transport-integration section, then converting to the temperature domain where the components associated with the latent-heat terms are integrated in time. Upon completion of the this step the updated temperature fields are converted back to the θ_c domain for further transport integration.

It is important to note that the model’s governing equations could be approximated without resorting to this operator-splitting process. The technique does, however, offer the advantages of computational simplicity and dimensional flexibility. This is the primary reason, for example, that the code can be adapted easily for service in one-, two-, or three-dimensional form.

3.2 Numerical Integration of Transport Terms

Spatial integrals of the advection and diffusion components of equation (2.10) are approximated using a Galerkin/finite-element approach with linear basis functions. This approach has been described elsewhere [e.g., Lapidus and Pinder (1982) and McRae, et al. (1982)] and will not be discussed at length here. In summary, the technique approximates spatial variations of the dependent variables as linear functions between each grid point, and then estimates appropriate coefficients of these "basis functions" by constraining the system to give the best possible fit, in a least-squares sense, over the total domain. Time derivatives of the advection and diffusion terms are approximated using standard Crank-Nicholson differencing, applying a technique similar to that described by Carmichael and Peters (1984).

3.3 Numerical Integration of Physicochemical Transformation Terms

The code framework is sufficiently flexible to accommodate a variety of numerical techniques for integration of the physicochemical transformation term of equation (2.10). The transformation-term integration procedure used to produce the example calculations of this user's manual is based on an exponentially-assisted/asymptotic approach, which has been described in detail in an earlier report (Easter and Hales 1984). This technique requires the segregation of production and decay terms, and approximates the decay rates as pseudo first-order decay processes. The resulting exponential equations give rise to the term "exponentially assisted", which is used often to describe the technique. The user has the option of applying the exponentials directly, or using Taylor's-series approximations. In the latter case the method is referred to as "asymptotic". This option has the possible advantage of increased computational efficiency, depending on the speed at which the machine at hand computes exponential functions.

3.4 Numerical Filtering Techniques

Under conditions where sharp spatial concentration gradients exist and advection dominates as a transport mechanism, especially where nonlinear phenomena occur, the composite numerical-integration technique can generate significant levels of high wave-number computational noise. To reduce this problem the nonlinear filtering method of Forester (1977) has been incorporated with the code. Typically interrogated after each one-dimensional transport step associated with equation (3.2), the code's filtering subroutines are written such that filter parameters can be selected individually for each dependent variable, with the option of no filtering whatsoever. Often filtering is not required, except for media having high fall velocities, such as rain under high precipitation rate conditions.

3.5 Performance of Numerical-Integration Technique

The Crank-Nicholson/Galerkin technique described above, in conjunction with the locally one-dimensional approximation, has been demonstrated to be second-order accurate in both space and in time (Lapidus and Pinder, 1982). The transport-integration component of the code was tested by allowing it to process several test patterns, including those documented by McRae, Goodin, and Seinfeld (1982). The present advection scheme, which is similar to the one described by these authors, produces almost identical results.

An older version of the exponentially-assisted/asymptotic transformation integrator has been applied previously with the Pluvius version 5.0 code (Easter and Hales, 1984), and its accuracy is discussed at length in that user's manual. The numerical method possesses the difficulty, shared by all exponentially assisted techniques, that it can diverge markedly from the true solution under specific circumstances; thus it is recommended that parallel preliminary computations be performed with this and a more robust, semi-implicit scheme for accuracy checking, prior to implementing any new physicochemical mechanisms with the code.

The most important advantages of the exponentially assisted integrator presented in this user's manual are its ability to deal effectively with stiff differential equations and its capability to operate with minimum historical data as it proceeds in time. This lowers demands on computer memory and time appreciably, and offsets the above disadvantages in most cases. As noted above, the composite code is sufficiently modular to permit substitution of an alternative ordinary differential-equation integrator when desired.

Section 4

Overview of Code Properties

The numerical techniques outlined in Section 3 permit substantial versatility to be incorporated with the code's architecture. In particular, the dimensionality can be adjusted to one-, two-, and three-space calculations by setting a few control variables, which confine the application of equation (2-10) to include z-terms only, z- and y-terms, or the full complement of z-, y-, and x-terms. Spatial grids may contain as many elements as desired by the user, within the constraints of computer resources. Variable grid spacing is possible, with essentially complete flexibility at the option of the user, although spatial configuration cannot be changed once the computations are in progress.

As noted above, temporal grid-spacing is segregated into two areas: the time-step associated with integration of transport terms and that required for integration of transformation processes. The transport time-step is totally adjustable and, once execution is initiated, it is controlled by a user-modifiable subroutine. Typically this subroutine limits the time step on the basis of a Courant-number criterion. The transformation time-step is computed internally on the basis of convergence between predictor and corrector calculations of the transformation integrator. As noted previously, this convergence is established by user-supplied accuracy specifications. The code operates such that the transformation time-step cannot exceed the 2Δt time-step for transport employed by equation (2.15). The final transformation time-step within any transport increment is adjusted automatically such that it fits exactly within this increment; thus transformation operations can be envisioned as being "embedded" within the computational framework for transport.

The modularity of the code allows all transport properties to be supplied by replaceable subroutines. These may take the form of calls for input from mass-storage devices if extensive amounts of data input are required. All wind and diffusivity components may vary with space and time in any prescribed manner. In this regard, it should be noted that the code accepts wind fields indiscriminantly, regardless of whether or not mass-conservation is preserved; thus the user is obliged to check for mass consistency using some independent method, before submitting a given wind field for computations.

The treatment of physicochemical transformation processes is also modular, and is identical to that employed by the code's Pluvius version 5.0 predecessor. This treatment is centered around the concept of "interaction diagrams" such as shown in Figure 4.1, which allow direct linkages of cloud-physics and chemical-reaction processes. To implement a set of transformation mechanisms within the code, one simply creates the appropriate set of interaction diagrams, writes rate or equilibrium expressions corresponding to each of the arrows, and codes these in the transformation subroutine. Once all required information is coded into its modules, the code can be executed directly to produce predicted concentration fields associated with all species-medium combinations.

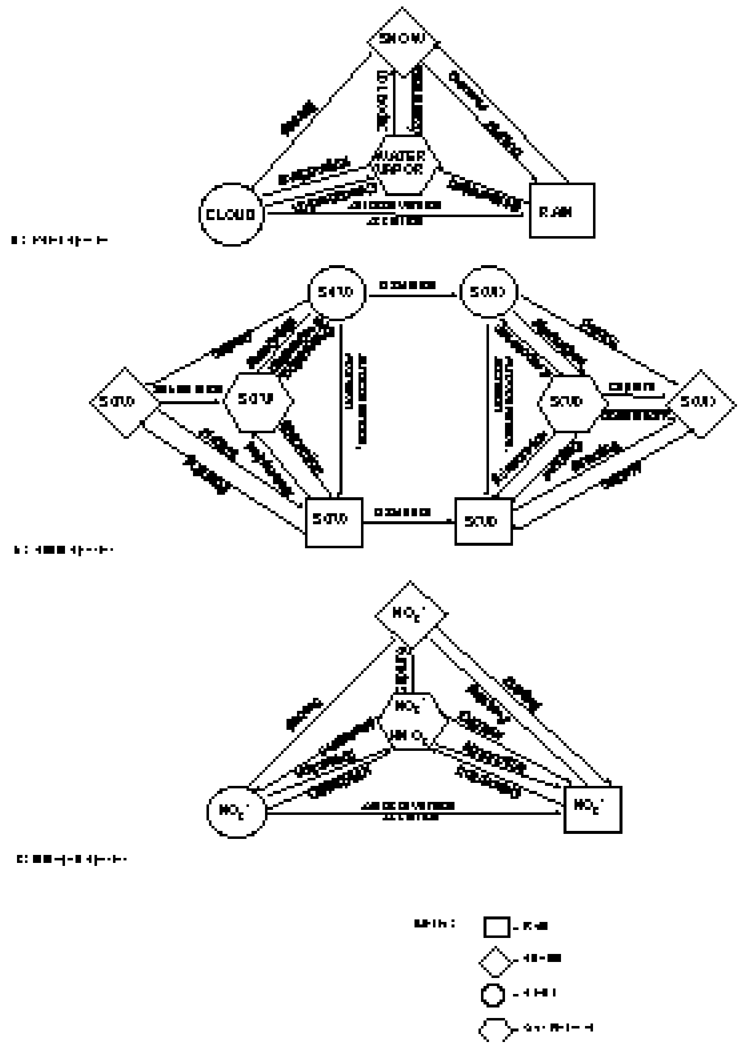

Figure 4.1: Example Interaction Diagrams for Code-Setup

Section 5

Example Computations

5.1 Description of Case Examples

In this section we provide a brief physical description of the two case examples that are used to illustrate the Pluvius II code in the next section. These examples involve two-dimensional simulations of chemistry and precipitation scavenging in a warm-frontal storm, adapted from Hales (1989). The storm depiction applied for this purpose is based on an idealized two-dimensional cross-section of a warm-front, with a time-invariant wind field specified as shown in Figure 5.1. The code was executed for vertical and horizontal grid-spacings of 150 meters and 13.75 km, respectively, on a two-dimensional domain 5.1 km high and 412.5 km long. Temporal grid spacing for integration of the horizontal transport processes was held constant at 7.5 seconds throughout the computations.

Code executions for this example were conducted in two phases, which will be referred to here as Case Examples A and B. Case Example A pertains to meteorological variables only, and execution of its associated code resulted in the cloud, snow, and rain fields that were applied for subsequent chemistry and scavenging calculations in Case Example B. Simulations for both case examples were continued out to 10 hours simulation time, a period which was sufficient to allow the system to approach steady-state conditions with reasonable accuracy.

Chemical simulations in Case Example B were performed by first reading the meteorological data at ten hours simulation time generated by Case Example A. These fields were held fixed during the succeeding chemical simulations which, as indicated previously, were carried out to a simulation time of ten hours as well.

Mechanistic approximations and physical assumptions associated with Case Examples A and B are summarized in Table 5.1. Entries in this table may be compared with the interaction-diagram paths in Figure 4.1. Any or all these somewhat limited characterizations may be easily improved or otherwise modified by the reader, simply by making the appropriate changes in the associated Pluvius II subroutines described in Section 6. Boundary conditions associated with the two case examples are summarized in Table 5.2.

The wind field applied for these simulations, shown Figure 5.1, was time-invariant with respect to the frontal position. Generated by a subroutine (subroutine **wind**) within the code, this field depicts an idealized warm front with warm air ascending from the left over the frontal surface. Cold air enters the system from the lower right, and reverses direction and ascends as it approaches the frontal surface from below.

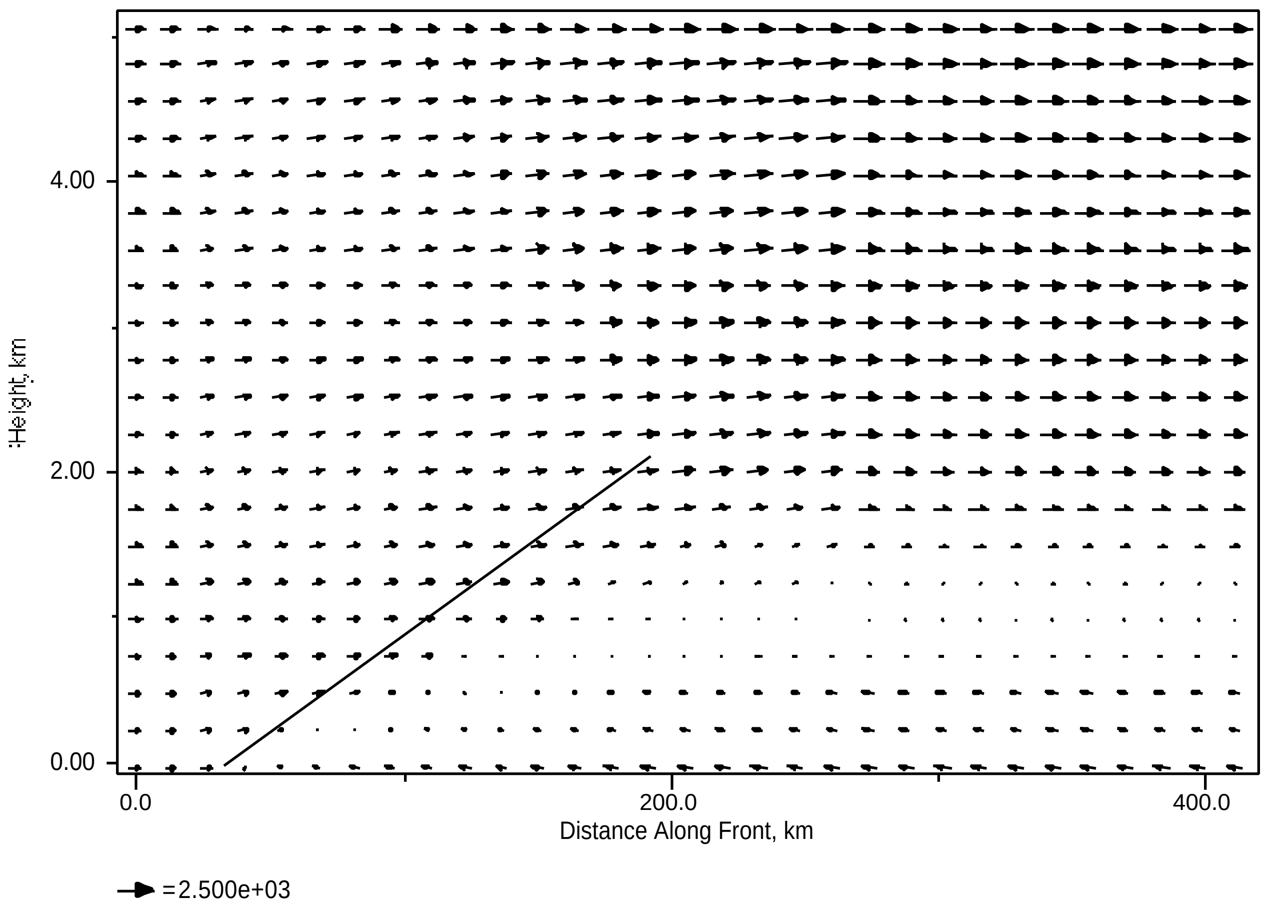

Figure 5.1: Plot of Wind Field Used for Example Simulations. Warm-frontal surface depicted by heavy line. Vector units in cm/s.

Because the intent of this user's manual is to instruct the user on operational aspects rather than to provide an exhaustive listing of results, we shall limit the examples of computed output here. Figure 5.2 shows selected meteorological fields computed by the Case Example A simulation at ten hours simulation time. One should note that the only transported variable shown here is rain. The others that appear in the figure (cloud water and temperature) are constrained variables, which are derived from the transported variables water vapor and equivalent potential temperature (not shown in Figure 5.2).

Figure 5.3 shows selected chemical fields computed by the Case Example B simulation at ten hours simulation time. Here only two variables are shown: the sum of sulfate aerosol and cloudborne sulfate ion, and rainborne sulfate ion. In contrast to the situation for Figure 5.2, both of these are transported variables.

Table 5.1: Summary of Physicochemical Parameterizations Applied for Case-Example Simulations A and B

Interaction	**Parameterization**
Cloud droplet autoconversion	Berry-Reinhardt (1974) parameterization.*
Cloud droplet accretion	Scott (1982) parameterization.*
Cloud droplet riming	Scott (1982) parameterization.*
Vapor deposition to ice	Easter-Hales (1984) parameterization.*
Raindrop evaporation	Kessler (1969) parameterization.*
Raindrop freezing	Lin et al. (1983) parameterization.*
Droplet nucleation (by both sulfate and nitrate particles)	Assumed first-order (in particle concentration) rate of 0.01 sec^{-1} in updraft; zero otherwise.
Aerosol capture by rain (both sulfate and nitrate particles)	Based on scavenging-coefficient relationship, taken from Dana and Hales (1976).
Sulfur dioxide solubility in water	Equations from Hales and Sutter (1973).
Ozone solubility in water	Implicitly incorporated in rate relationship of Penkett et al. (1979).
Nitric acid solubility in water	Relationship of Schwartz and White (1981).
Hydrogen peroxide solubility in water	Relationship of Martin and Damschen (1981).
Ozone mass transfer to rain	Solubility equilibrium assumed.
Sulfur dioxide, nitric acid, and hydrogen peroxide mass transfer to rain	Reversible mass transfer to raindrop ensemble; generalization of Froessling equation [Bird, et al., (1960)]
Ozone oxidation of S(IV) in cloud water	Solubility equilibrium assumed for ozone and sulfur dioxide; Expression of Penkett et al. (1979) applied to describe oxidation rate.
Hydrogen peroxide oxidation of S(IV) in cloud water	Solubility equilibrium assumed for sulfur dioxide and reversible mass-transfer theory applied for hydrogen peroxide. Expression of Penkett et al. (1979) applied to describe oxidation rate
Hydrogen peroxide oxidation of S(IV) in rain water	Reversible mass-transfer theory applied for both sulfur dioxide and hydrogen peroxide. Expression of Martin and Damschen (1981) applied to describe oxidation rate.
Hydrogen-ion concentration in cloud and rain water	$[H^+] = 1.03X10^{-5} + 1.18[SO_4^{2-}] + 0.767[NO_3^-]$ (Based on composite behavior observed on the MAP_3S precipitation-chemistry network [MAP3S (1982)].

*Cloud-physics treatment identical to that described by Easter and Hales (1984).

Table 5.2: Boundary Conditions Applied for Case-Example Simulations A and B

Dependent Variable	**Boundary Condition**
Temperature (used to derive θ_c)	Surface temperatures: 290°K on warm side and 282°K on the cold side. Adiabatic profiles imposed.
Moisture (vapor + cloud water)	65%RH up to 2850 m on warm side and 45%RH on cold side; decreasing aloft as exp(20-z/150).
Sulfur dioxide + S(IV) in cloud water	r(0) = 10 ppb. Flux computed by formulas 1 and 2.
Sulfate aerosol + sulfate in cloud water	r(0) = 1 ppb. Flux computed by formulas 1 and 2.
Sulfate aerosol	Computed above as for combined species. Cloud water does not exist in inflow regions.
Nitric acid + nitrate in cloud water	r(0) = 1 ppb. Flux computed by formulas 1 and 2.
Nitrate aerosol	r(0) = 0.2 ppb. Flux computed by formulas 1 and 2.
Hydrogen peroxide	r(z) = 1 ppb. Flux computed by formula 1.
Ozone	r(z) = 50 ppb. Flux computed by formula 1.

Notes:

All other dependent variables set to zero at inflow boundaries.

Formula 1:

Flux (z) = r(z)v(z)c where v(z) = local horizontal velocity.

Formula 2:

r(z) = r(0) between z = 0 and z = 1500 m;

r(z) = r(0) exp[0.0001(1500-z) above z = 1500 m,

where r is the molar mixing ratio and r(0) is its value at z = 0.

5.2 Executing the Case Example Codes

To reproduce the Case Example A and B simulations, the user should apply the following procedure:

1. Compile and link the total code using versions of the include file `pluvius.com` and the subroutines `inptgn`, `init`, `cleanup`, and `gen` that are appropriate to Case Example A. These are listed in Tables 6.3a, 6.4a, 6.6a, 6.13a, and 6.43a.

2. Execute the resulting executable code to produce the output files **diskoutpt** and **diskbkup**. (Upon execution, the user will observe screen output at each time-step corresponding to temperature and humidity data at grid index **j=15,k=19**. The variable **diff** represents the difference between the saturation humidity and the computed sum of vapor plus cloud water; it will approach zero as the local air becomes saturated and go positive after cloud formation occurs. The print statements producing this screen output are located in the main program **pluvius** and should be removed after the user gains sufficient confidence in the code.) The code will terminate at 10 hours simulation time (36000 seconds).

3. Use the meteorological fields contained in file **diskoutpt** to compare with those in Figure 5.2.

4. Prepare for the Case Example B simulation: Delete or rename file **diskoutpt**. Rename the binary file **diskbkup** (which was produced by the Case Example A simulation) to **diskbkup.exa**, so that it will be recognized by the input subroutine **cloudinit**. Compile and link the total code using versions of the include file **pluvius.com** and the subroutines **inptgn, init, cleanup,** and **gen** that are appropriate to Case Example B. These are listed in Tables 6.3b, 6.4b, 6.6b, 6.13b, and 6.43b.

5. Execute the resulting executable code to produce the new output file **diskoutpt**.

6. Use the chemical fields contained in file **diskoutpt** to compare with those in Figure 5.3.

The above operations are straightforward, and will require three to four hours to complete on a typical work station. Once these are completed successfully the user should experiment with the code further by modifying various model elements, such as boundary conditions, chemical species, or physicochemical mechanisms. Subsequent to this the user should have sufficient confidence to make more substantial changes to the code and to apply it for more complex physical and chemical environments.

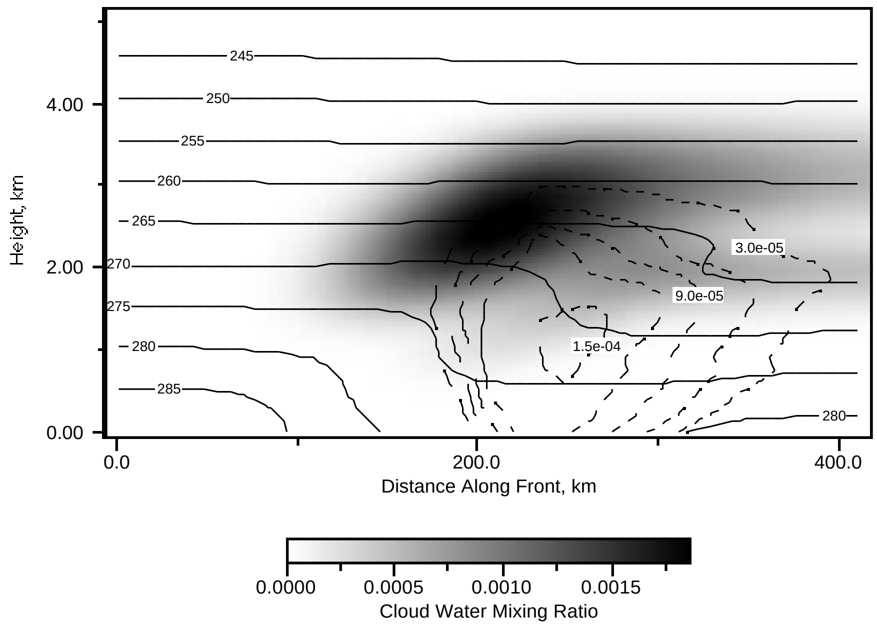

Figure 5.2: Selected Results of Case Example A Simulation at 10 Hours Simulation Time. Solid contours depict temperature in degrees K, and dashed contours depict rain mixing ratios. Shading depicts cloud-water mixing ratios. Mixing-ratio units are moles water per mole air.

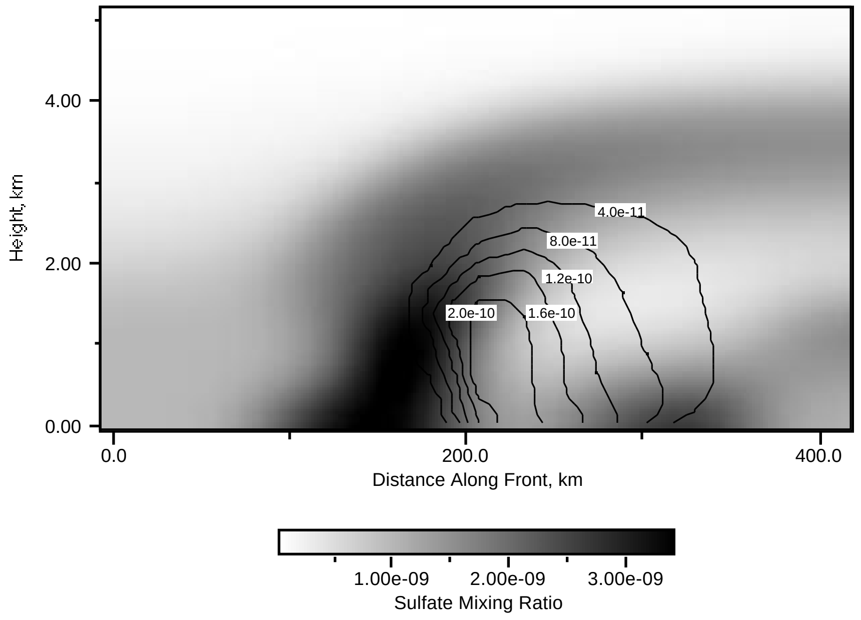

Figure 5.3: Selected Results of Case Example B Simulation at 10 Hours Simulation Time. Solid contours depict rainborne sulfate-ion mixing ratios. Shading depicts aerosol + cloudborne sulfate mixing ratio. Mixing ratio units are moles sulfate ion per mole of air.

Section 6

Detailed Code Description

6.1 Introduction

This section provides a detailed description of the Pluvius II code in a logical progression which follows, at various levels of depth, the code’s execution flow. Considerable amounts of material are presented here, and we recommend that persons unfamiliar with Pluvius II do not attempt to read through this section from beginning to end in a completely linear fashion. We suggest, rather, that the new reader peruse down through the first few subroutines to get an initial idea of the logical flow and of how input is submitted to the code, to the depth of the core subroutine that controls the numerical integration sequence (named subroutine **`core`,** not surprisingly). Following this initial perusal, the unfamiliar reader is advised to immediately follow the procedure indicated in Section 5 to execute the demonstration program on his or her own machine. Subsequently, the user can examine various aspects of the code by making progressive changes in selected subroutines, such as modifying wind fields, changing kinetics mechanisms and parameters, and so-forth, until sufficient confidence is acquired to move directly to the user’s intended application. Definitions of the code's common-block variables are summarized in Table 6.1.

A few additional initial comments are helpful at this point. First, it is useful to note that Pluvius II’s preferred unit system is cgs, and that the code’s dependent variables (except for temperature) are normally expressed in terms of mixing ratios (moles pollutant per mole of air). Other units are employed on occasion for internal expediency, but this practice is limited inasmuch as is practical.

Second, the description given here follows the two Case Examples of this user’s manual, which were described previously in Section 5. While many of the code’s subroutines are invariant with respect to any specific application, others are intended to be modified by the user to fit specific purposes at-hand. Consequently, several of the subroutines appearing in the following pages are presented twice — once for Case Example A and once for Case Example B — and each subroutine is relegated to one of three classes as follows:

> <u>Class I Subroutines</u> - Those subroutines that are essential for operation of the code, but are intended for modular replacement by the user.
>
> <u>Class II Subroutines</u> - Those subroutines that are essential for operation of the code and in general should be left unchanged.

Class III Subroutines - Those special-purpose subroutines, to be supplied by the user, that are necessary only for the specific modelling purpose at hand.

This approach to the presentation is intended to provide the user with additional insight regarding further modifications of the code for his or her specific application.

6.2 Overview of Main Program and First-Level Subroutines

The code's main program, **`pluvius,`** is shown in Table 6.2. As its internal documentation suggests, this main program first calls subroutine **`inptgn`** to define the vertical and horizontal grids as well as to read in control variables and all other associated input except for that pertaining to the code's initial conditions. It then calls subroutine **`femset,`** which defines several arrays used for the finite-element transport integrations. Subsequently the code sets the model's initial conditions (subroutine **`init`**), and then performs an inflow/outflow material balance over the computational domain by first establishing the initial wind field (subroutine **`wind`**) and then calling subroutine **`matbal.`** It should be noted in this context that execution of the material-balance subroutine is something of a bookkeeping exercise and is not an essential activity of the code; but overall material-balance information is often a highly desirable by-product of the base computations. During this initial process the code also writes to disk storage the initial concentration fields by calling subroutine **`printi.`**

Next, the execution proceeds to the code's main computation loop. Subroutine **`wind`** is called once again (this is redundant for the first pass through the loop, since the wind fields for this example are time-invariant and have been computed in the previous call to **`wind.`** Typical versions of **`wind`** sense this redundant call under such conditions and simply perform a no-action return). Diffusivities are then calculated for each point in the computation domain (subroutine **`diff`**), and the execution proceeds to subroutine **`core`** which, as indicated above, orchestrates the numerical integration process for one composite time-step.

Subroutine **`cleanup`** is then called. This subroutine corrects for the propensity of the numerical methods called by subroutine **`core`** to produce small, but still nonzero, values of the dependent variables in regions where, by definition, values should be identically equal to zero. A prime example of this is the region in clear air, where residual small values for the concentrations of chemical species in cloud water are often generated by the code. These numbers are very small and thus pose no real problem, but the output's readability is enhanced significantly if they are replaced by zero values prior to printing.

Depending on the print-control variables, the computed results may be written to disk files at this point. Typically two different disk files are written for this purpose. The primary disk file is named **`diskoutpt`**, assigned to unit 20, and created as an ASCII file by subroutine **`printi`**. It is intended for outputting the conventional computed results of the code. A second disk file is named **`diskbkup`**, assigned to unit 21, and created as an unformatted file by subroutine **`collect`**. It contains all of those variables that would be necessary to restart the simulation at the current model time, **`t`**. This file is used mainly for subsequently restarting the computations in the event of some system interrupt. Upon writing the appropriate results, subroutine **`matbal`** is called again to produce the material-balance values for the total computational domain. At this point — and provided the time-limit **`tstop`** has not been exceeded — the code repeats the main computation loop for a subsequent time-step. If **`tstop`** has been exceeded, the code executes a normal termination and concludes the active program.

In conclusion to this cursory overview of Pluvius II's main program, it should be emphasized that several of the first-level subroutines described here (**`inptgn`**, **`init`**, **`wind`**, **`printi`**, **`print`**, **`diff`**, **`cleanup`**, and **`matbal`**) are application-specific and thus are intended to be modified by the user. One should note also that most of the code's variables are communicated between subroutines via common blocks in the include file **`pluvius.com`**. The **`pluvius.com`** files for the two case examples described in this user's manual are given in Tables 6.3a and 6.3b.

Versions of subroutine **`inptgn`** corresponding to Case Examples A and B are shown, respectively, in Tables 6.4a and 6.4b. In Case Example A the procedure is relatively straightforward, with the pertinent information being entered via the subroutine in an essentially linear manner. First, the **`mtrans`**, **`mtrany`**, and **`mtranx`** variables are set for a two-dimensional execution, allowing transport calculations to be performed in the vertical- and y-directions only. Next a two-dimensional grid is established, which contains 31 and 35 grid points in the horizontal and vertical dimensions, respectively.

This Case-Example A simulation includes four transported variables (the sum of cloud water and water vapor, rain, snow, and equivalent potential temperature) plus three constrained variables (cloud water, water vapor, and temperature), and the code is informed of these quantities by setting the variables **`ltot`** and **`ltot2`**. Vapor and cloud water are treated as a single summed variable here for two reasons. First, this practice eliminates abrupt discontinuities in the transported variables at the cloud boundaries; second, when equivalent potential temperature is used as an advected thermodynamic variable, both water vapor and cloud water become constrained by the zero-supersaturation assumption.

As noted previously, Pluvius II allows different transported variables to move in different fashions, depending on their physical properties; thus snow and rain, having unique fall velocities, are allowed here to advect in individual "regimes," which in turn are different from those of the gaseous species and temperature. The total numbers of transport regimes for a particular simulation are specified by the variables **`lpxtot,`** **`lpytot`** and **`lpstot.`** In this particular example four vertical (snow, rain, air, and temperature) and two horizontal (air and temperature) transport regimes are specified. An individual regime is allocated here for temperature (heat transfer), presuming that the eddy diffusivity for this case may be different than that for the more convectively passive species.[1]

Next, integer index variables and alphanumeric species names are associated with the numerical index-values for each of the transported and constrained variables. The index variables are primarily for recognition convenience when the indices are used in later portions of the code. The alphanumeric species names, in combination with a run title which is also supplied at this point, are used in subsequent disk output of computed results. An essential rule in the naming operation is that indices of the transported variables must range from 1 to **`ltot`** and those of the constrained variables must range from **`ltot + 1`** to **`ltot2;`** otherwise, species indices may be set in a totally arbitrary manner. Transport-regime indices are then assigned to the arrays **`lprops`** and **`lpropy.`**

Next, the code's output controls are established (the subroutine **`restart`** is bypassed in this Case Example). **`nprint`** is set to write to a primary disk file after every 480 composite integration time-steps (every 7200 seconds). **`itdisk`** is set to write updated records to a second, backup disk file, erasing the previous record upon activation. **`tdisk`** and **`dtdisk`** are set to start writing the backup records after one hour of simulation time and to repeat every hour thereafter.

Following this, the code's primary temporal and spatial grid-spacing parameters are established. The simulation's start time is initialized to zero, the advective time-steps are set to 7.5 seconds, and the simulation time limit is set at ten hours. **`dsunif`** and **`dyunif`** establish the z and y grid spacings in centimeters and instruct the code to apply uniform vertical and horizontal grid spacings, and **`scord(1)`** and **`ycord(1)`** set the lower positions of the computational grid to zero. Next, subroutine **`grdxyz,`** shown in Table 6.19, is called to set up the computation grid. For uniform-grid conditions **`grdxyz`** simply sets the arrays **`xcord,`** **`ycord,`** and/or **`scord`** as appropriate, and returns to subroutine **`inptgn.`** If a nonuniform grid is indeed desired, then grid data in the form

[1]The diffusivity fields used for heat and mass transfer are identical in Case Example A . The extra regime provided here is simply illustrate the availability of this option, if different diffusivity fields are desired.

of **xcord, ycord,** and/or **scord** arrays should be entered explicitly through subroutine **inptgn** prior to the call to **grdxyz.**

Next, subroutine **inptgn** proceeds to identify control variables for execution of the ordinary differential-equation (ODE) integrator **odeint.** These variables are specific to this particular integrator, and must be appropriately replaced if an alternate integrator is applied. **eps s**ets an error limit for the ODE code, and **dtmin** establishes the smallest time-increment allowed before the code exits with an error return. **ncorr** and **iexp** are set, respectively, so that a single correction pass is applied in the code's predictor-corrector cycle, and true exponentials (as opposed to Taylor's-series approximations) are applied to simulate species decay rates. **ratesf** and **cmin** establish, respectively, the species decay rates beyond which stiff ODE calculations are applied and the concentrations below which the associated variables are considered identically equal to zero. **cerror** is an array containing values of the dependent variables below which relaxed convergence criteria are applied. Most of these variables can be considered tuning parameters which are employed to optimize performance of **odeint.** The ones given here result in acceptable code operation for present purposes, but the user can adjust these as he or she feels appropriate.

Several of the following variables are obvious from the code listing. The boundary conditions are set to flux-specified types by setting the lower and upper boundary-condition specifier variables to unity for each of the transported species. The subroutine frequency variables are set so that transport properties are updated with each advective time-step and several control-toggle variables are set: **mgen = 1** sets the code so that its ODE portion is operative; **mtemp = 1** instructs the code to include an energy balance as part of its calculation; and **nonsym = -1** tells the **core** subroutine to operate in its normal, "symmetric" operational mode.

Next, subroutine **inptgn** sets the control parameters for the code's filtering scheme, which is used to reduce high-frequency noise produced by the numerical integration process. Without going into large detail at this point, **nfilts** and **nfilty** define the number of advective time steps prior to each filter application in the respective directions. **nfilsrch, nfiltset,** and **nfiltitr** are internal parameters used by the filtering scheme, and **filfacs** is an array describing the degree of filtering for each transported species (these should be set to 0.0 for no filtering whatsoever). Further information on Pluvius II's filtering scheme can be acquired from the listing for subroutine **foresfilt,** which is given in Table 6.40 and by referring to Forester (1977). The final operation of subroutine **inptgn** is to turn off the boundary filter for the x-direction, and to turn on its counterparts for the vertical- and y-directions. Setting the last two variables to unity activates additional filtering of results in the boundary regions.

The structure of subroutine **`inptgn`** for Case Example B is quite similar to that for Case Example A. As with **`lvapcl`** in Case Example A, some of the transported chemical species have been summed for computational convenience; thus **`lso2gcl`** represents the sulfur dioxide that occurs in the gas phase plus that which is dissolved in local cloud water, and the remaining variables take on their obvious significances (constrained variables **`lhcloud`** and **`lhrain`** pertain respectively to the hydrogen-ion content of cloud- and rain-water, and are calculated simply from an ion balance). Case Example B includes 15 transported species (ending with gas-phase ozone). The variables such as **`lrain, lsnow,`** and **`lvapcl`** are now relegated to "constrained" status, since they are specified here from the results of Case Example A and are not computed as a part of Case Example B's numerical-integration scheme.

Final portions of this **`inptgn`** version are straightforward extensions of that in Case Example A. The only feature that deserves some additional comment in this context is the transport-regime designation for the various chemical species, noting that chemicals associated with falling rain and snow inherit the terminal fall speeds of these meteorological entities, and thus are assigned the associated values of **`lprops.`**

Subroutine **`femset,`** which establishes key grid-related parameters for the finite-element integration, is shown in Table 6.5. The first task performed by this subroutine is to zero a number of the finite-element integration parameters, having the various variable names **`ct—(idum,i),`** where the index **`idum`** pertains to dimension x, y, or z (1,2, or 3, respectively) and **`i`** is the associated grid index. It then computes the associated values of the finite-element integration parameters for all dimensions that are active in the simulation.

Subroutine **`init`** assumes final responsibility for establishing the initial conditions for the transported variables **`r(i,j,k,l)`** for the total computational domain. For Case Example A, this involves stipulating surface pressures and temperatures for the warm and cold sectors, deriving temperature and humidity profiles for associated air columns on the basis of adiabatic lapse rates and specified relative humidities, and then applying these profiles to warm- and cold-sector portions of the computational domain. The code in Table 6.6a begins to the left of the front, in the warm-sector air, and stipulates a surface temperature of 290 °K for this region. Next it computes a corresponding air concentration using an equation of state and sets a water-vapor mixing ratio that is 65 percent of its saturation value (here **`sat`** is an internal function that returns a concentration of water vapor in saturated air, when supplied the local absolute temperature).

Following these and some additional bookkeeping steps, the code estimates the concentration of air at points aloft using an approximate hydrostatic equation. The resulting air concentrations then are applied to compute temperatures aloft, which in turn are used to set water-vapor mixing ratios such that relative humidities are 65% up to the 20th grid point and decrease aloft. Subsequently, these initial temperature, pressure, mixing-ratio, and concentration profiles are applied to all internal grid points.

This general procedure is then applied for cold-sector air to the right of the front. Here, however, the surface temperature is set to 282 °K and the relative humidity is set at 45 percent. The associated temperature, pressure, mixing ratio, and concentration profiles are subsequently "painted" into the cold-wedge region below the frontal surface, resulting in a temperature and humidity discontinuity across the front.

One should note that no clouds occur in this initial environment, and that we should expect clouds to emerge in the simulation as the moist air moves upward over the frontal surface. Also one should note that, since the code's execution will progress to simulation times that are sufficient to render the system essentially boundary-condition limited, any reasonable set of initial conditions would suffice for present purposes: the ones described here were chosen simply because they give an uncomplicated but reasonable representation of behavior, which is sufficient for initiating the computational process.

After initial conditions have been set, subroutine **`inflowinit`** is called, which stores initial boundary concentrations for the subsequent calculation of inflow boundary conditions. The final action of subroutine **`init`** is to call subroutine **`restart`** if **`initmode`** specifies a restart from a previous backup disk file. As noted in the documentation, the current version of **`inflowinit`** is intended for time-invariant boundary conditions only, and the user should modify (or possibly eliminate) this subroutine if time-variant boundary conditions are imposed.

The version of subroutine **`init`** for Case Example B is simpler than its Case Example A counterpart, mainly because the meteorological variables have been preset by Case Example A's execution, and need only to be copied here. This operation is performed by calling subroutine **`cloudinit,`** which reads the unformatted disk file that was written by the Case Example A execution and subsequently renamed **`diskoutpt.exa.`**

The Case Example B initial conditions for the chemical variables have been chosen to be rather simple in form: constant mixing ratios up to 1500 meters and decreasing aloft, except for hydrogen peroxide and ozone. As with Case Example A, we expect this chemical system to be boundary-condition limited at large simulation times; thus any reasonable set of initial conditions should suffice here as well.

Subroutine **`wind`** establishes numerical values for the x, y, and z components of the model's wind fields. The example of subroutine **`wind`** given in Table 6.7 sets the two-dimensional wind field used for the two-dimensional Case Examples A and B. Since these examples apply a wind field that is fixed in time, the code appearing in Table 6.7 implements an immediate no-action return after its initial interrogation.

The wind field described by this subroutine version corresponds to the vertical slice of an idealized warm-frontal system that was shown in Figure 5.1. Winds within this scheme are relative to the motion of a warm-frontal surface, which progresses from left to right, with air from the warm-frontal zone moving to the right and upward over the frontal surface. Cold air at low elevations below the frontal surface moves to the left (relative to the front), ascends, and then reverses direction at higher altitudes.

The subroutine calculates mass-consistent wind fields for four specific zones of Figure 5.1. The first of these is to the left of the surface front: here the vertical velocities are zero, and the horizontal velocities scale with air density (for mass-consistency) in a straightforward manner. The second zone occurs in the region above the ascending warm front. Here vertical velocities in the vicinity of the frontal surface correspond to movement over this surface, and fall to zero at the top of the model domain. Again, the code adjusts the horizontal winds to achieve mass consistency in reflection of the prescribed vertical motions.

The third zone is the cold-air wedge underlying the cold front. This portion of the field is calculated by allowing the horizontal winds immediately below the frontal surface to have horizontal and vertical components which are identical to those of the air immediately above the front, and then computing material fluxes at lower grid points necessary to maintain mass-continuity in this region. Finally, velocity components for the fourth zone (the air immediately to the right of the front) are computed in a manner that preserves mass consistency and harmony with the wind-fields in the adjoining frontal region.

The code’s utility printing subroutines, **`printi`** and **`print,`** are listed in Tables 6.8 and 6.9, respectively. As can be noted from the main **`pluvius`** code in Table 6.2, **`printi`** is called once only, and is intended to output any code diagnostics (to file **`diagnostics`**) and the initial concentration and temperature fields (to file **`diskoutpt`**, which is intended for subsequent use in outputting corresponding fields computed at future simulation times).

Subroutine **`print`** is quite similar to **`printi.`** In contrast to the initial printing routine, however, this subroutine does not open the output files (since they have been opened previously). The examples of **`printi`** and **`print`** shown here have been used to output the primary numerical results for Case Examples A and B. Since the desired output formats will be strongly dependent on the modeling application at-hand, both **`printi`** and **`print`** are intended for modification by the user.

As noted above, subroutine **`matbal`** (Table 6.10) is essentially a bookkeeping routine, which performs integral material balances over the computational domain to determine advective inflow/outflow rates and material fluxes. **`matbal'`**s basic approach is to operate sequentially on the domain's left-hand, right-hand, and then bottom boundaries. At each boundary it computes advective fluxes at all appropriate grid points, integrates these spatially to obtain total material-balance components, and then prints the resulting data to disk. Subroutine **`integrate`** is a simple quadrature routine which is applied for this purpose. In the case of the lower boundary, the vertical fluxes are computed from the vertical transport velocities, which result from sedimentation of the pollutant-containing precipitation. These velocities are calculated using subroutine **`transport,`** and are stored in the array **`vertcclm.`** One should note here that snow does not reach the ground in this simulation and that dry deposition is ignored; thus the only deposition mechanism is through rainfall. The subroutine should be modified by the user in situations where these conditions are not met. One should note here again that subroutine **`matbal`** is not necessary for execution of the Pluvius II code, and can be deleted from the main program, if desired.

Subroutine **`diff,`** shown in Table 6.11, sets diffusivities for each transport regime at each grid point of the computational domain. The very simple example shown here establishes all diffusivities at 10^5 cm^2/s. Since these are held constant with time, this version of subroutine **`diff`** executes a no-action return after its initial interrogation.

Subroutine **`core,`** shown in Table 6.12, is in many respects the heart of the Pluvius II code. In a typical execution this routine solves equation (3.2) for a double time increment 2Δt, in the sequence indicated by that equation:

$$\Delta\zeta\Big|_{\substack{\text{total,}\\ 2\Delta t}} = \Delta\zeta\Big|_{\substack{x,\\ \Delta t}} + \Delta\zeta\Big|_{\substack{y,\\ \Delta t}} + \Delta\zeta\Big|_{\substack{z,\\ \Delta t}} + \Delta\zeta\Big|_{\substack{\text{transformation,}\\ 2\Delta t}} + \Delta\zeta\Big|_{\substack{z,\\ \Delta t}} + \Delta\zeta\Big|_{\substack{y,\\ \Delta t}} + \Delta\zeta\Big|_{\substack{x,\\ \Delta t}} . \qquad (3.2)$$

In the event that the x- and or y-dimensions are inactive in a particular simulation, the code simply bypasses the corresponding steps. Pluvius II also gives the option of proceeding through equation (3.2) halfway, and thus computing for a single time-increment Δt. This option is controlled via the toggle-variable **`nonsym,`** as indicated in the code listing.

Subroutine **core00,** shown in Table 6.18, is called only on the first execution of subroutine **core.** It computes physical-property related variables which are time-invariant, and thus need to be computed only once during the simulation. After passing this point, the code next executes initial integrations for a single time-step in the x- and y-directions (as appropriate to the problem at-hand), calling the integration routines **xinteg** and **yinteg.** It is important to note in this context that a single interrogation of each of these subroutines results in integrations for all **ltot** transported species.

Subsequent to these operations, the code proceeds to the vertical-integration stage, where the process is somewhat more complicated for two reasons. First, and as mentioned previously, the current code version segregates vertical integration operations on a column-by-column basis. Within this operation, transported variables and pertinent other parameters are "packed" into special one-dimensional arrays by executing the subroutine **ijpack,** which is exhibited in Table 6.55. The second complication results from the frequent need, when precipitation exists, to reduce the time-step for vertical integration in order to preserve Courant-number stability. This is done by calling subroutine **sbstep,** which returns the (possibly) modified time increment **dtsub.** The listing for subroutine **sbstep** is shown in Table 6.54.

Upon accomplishing these initial steps for the vertical integration, subroutine **core** proceeds to process multiple **dtsub** increments, performing the operations

$$\Delta\zeta\Big|_{\substack{z,\\ \text{dtsub}}} + \Delta\zeta\Big|_{\substack{\text{transformation,}\\ \text{2dtsub}}} + \Delta\zeta\Big|_{\substack{z,\\ \text{dtsub}}}$$

repeatedly, until a full double increment 2Δt is traversed. Subroutines **sinteg** and **chmint** respectively coordinate individual integrations for the vertical dimension and for transformation. Upon completing this full cycle for all **i,j** columns, the code calls subroutine **sfilter** to conduct some required bookkeeping operations, and converts the columnized data back to its original **r(i,j,k,l)** configuration by calling subroutine **ijunpack.** A listing of **ijunpack** is given in Table 6.56.

Finally (and if symmetric integrations are desired), the code conducts further integrations for a single time-step in the x- and y-directions, again calling the integration routines **xinteg** and **yinteg.**

Versions of subroutine **cleanup** corresponding to Case Examples A and B are shown in Tables 6.13a and 6.13b, respectively. These subroutines zero-out variables under conditions where concentrations should be identically equal to zero, and also compute values of constrained variables as appropriate.

The computational procedures are generally obvious from the code listings, and thus little comment is necessary at this point. In the Case Example A version, temperature (a constrained variable) is calculated from the equivalent potential temperature (a transported variable) using subroutine **`tfrmth,`** which is listed in Table 6.51.

Subroutine **`collect,`** shown in Table 6.14, combines with **`printi, matbal,`** and **`print`** to provide an additional data-output utility. In contrast to **`printi`** and **`print,`** however, this subroutine is intended to log current computed results for the event that the user may desire to restart the code at an advanced time, subsequent to previous simulations. This convenience can be helpful under the contingency conditions of a system crash, or if the user desires to extend a simulation or perform it in pieces.

As can be noted from the code, the salient output data are written in unformatted form on disk file **`diskbkup.`** Through the control variables **`itdisk, tdisk,`** and **`dtdisk`** the user has the choice to record sequential time-frames of data, only the final time frame, or none at all. Subroutine **`collect`** is intended to serve as a generic subroutine and generally does not require modification by the user.

6.3 Overview of Second- and Lower-Level Subroutines

This subsection presents all of those subroutines within the Pluvius II code that are not called directly by the main program. In this presentation we shall attempt to follow a logical flow, starting with those subroutines mentioned in the above discussion, and proceeding through the total set in a manner that approximates the flow of the program's execution.

6.3.1 Initialization and Grid-Setup Operations

Subroutine **`restart,`** shown in Table 6.15, is responsible for reading the disk output generated by subroutine **`collect.`** This listing essentially mirrors that of subroutine **`collect.`** One should note here that subroutine **`restart`** is not used in the two case examples presented here. Rather, the file generated by subroutine **`collect`** in Case Example A is read in Case Example B by **`cloudinit`**, (Table 6.16) which was written as an ad hoc subroutine for this special purpose.

As noted above, subroutine **`inflowinit`** (Table 6.17) is called from subroutine **`init`** to preserve boundary concentrations for possible later use in computing fluxes for inflow boundary conditions. Also as noted above, subroutine **`core00`** (Table 6.18) is called from subroutine **`core`** (Table 6.12) once only, on the initial call to **`core.`** **`core00`** produces initial values of essential physical properties, by interrogating subroutines **`propsx`** and **`propsy`** when these properties are time-invariant and only need to be calculated once during a simulation. These property-generation subroutines appear in Tables 6.27 and 6.32, below.

Subroutine **`grdxyz,`** shown in Table 6.19, was discussed previously in the context of subroutine **`inptgn.`** This subroutine is called to establish effective grid spacings **`xwidth,`** **`ywidth,`** and/or **`swidth`** associated with each point of the computational framework. **`grdxyz`** also establishes the arrays **`xcord,`** **`ycord,`** and/or **`scord,`** as appropriate.

6.3.2 Transport-Integration Operations

Subroutine **`sinteg,`** shown in Table 6.20, is called by subroutine **`core`** to integrate the vertical components of the transported variables over one advective time-step, Δt. On the basis of equation (2.10) one can derive the following form for the vertical component of the mass-conservation equation, which is the essential equation for this vertical-component integration:

$$\left.\frac{\partial \zeta}{\partial t}\right|_z = \frac{\partial}{\partial z}\left(A_z \frac{\partial \zeta}{\partial z}\right) + B_z \frac{\partial \zeta}{\partial z} + C_z \zeta \,, \tag{6.1}$$

where

$$\left.\begin{aligned} A_z &= K_z; \\ B_z &= -w - w_m + \frac{K_z}{c}\frac{\partial c}{\partial z}; \\ C_z &= -\frac{w_m}{c}\frac{\partial c}{\partial z} - \frac{\partial w_m}{\partial z}. \end{aligned}\right\} \tag{6.2}$$

Here K_z and w represent the vertical diffusivity and wind-velocity components, and the remaining variables are defined as in equations (2.2), (2.10) and (3.2).

The forms for the horizontal directions are similar to equation (6.1), but do not contain the w_m term. The Galerkin finite-element approach applied by Pluvius II to each of these equations approximates ζ in equation (6.1) by the summation

$$\zeta(t,z) \approx q(t,z) = \sum_k q_k(t)\varphi_k(z), \tag{6.3}$$

where the q_k are the values of the approximation q at each grid-point k and the "basis functions" φ_k are linear functions that have values of unity at the grid points z_k, zero at immediate neighbor points, and zero at all locations beyond the immediate neighbors:

$$\varphi_k(z) = \begin{cases} 0 & \text{for } z \le z_{k-1} \text{ and } z \ge z_{k+1} \\ \dfrac{z - z_{k-1}}{z_k - z_{k-1}} & \text{for } z_{k-1} \le z \le z_k \\ \dfrac{z - z_{k+1}}{z_k - z_{k+1}} & \text{for } z_k \le z \le z_{k+1} \end{cases} \tag{6.4}$$

The spatial discretization of equation (6.1) is obtained by requiring that

$$\int \varphi_k(z)\left[\frac{\partial q}{\partial t} - \frac{\partial}{\partial z}(A_z \frac{\partial q}{\partial z}) - B_z \frac{\partial q}{\partial z} - C_z q\right] dz = 0 \tag{6.5}$$

for all k, where the integration limits correspond to the extent of the z-domain and where the coefficients A_z, B_z, and C_z as well as q are represented in terms of basis-function expansions. Evaluation of equation (6.5) is straightforward because of the simple form of φ_k and yields

$$f_k \frac{\partial q_{k-1}}{\partial t} + g_k \frac{\partial q_k}{\partial t} + h_k \frac{\partial q_{k+1}}{\partial t} = \alpha_k q_{k-1} + \beta_k q_k + \gamma_k q_{k+1} \tag{6.6}$$

where α_k, β_k, and γ_k depend on A_z, B_z, and C_z and thus are time-dependent, while f_k, g_k, and h_k depend only on the vertical grid spacing.

Crank-Nicholson time discretization is now applied to equation (6.6). Terms on the left-hand side are approximated as

$$\frac{\partial q_k}{\partial t} \approx \frac{q_k^* - \tilde{q}_k}{\Delta t} \tag{6.7}$$

where $q_k^* = q_k(t + \Delta t)$ and $\tilde{q}_k = q_k(t)$. Terms on the right-hand side are approximated as

$$\beta_k q_k \approx 0.5\tilde{\beta}_k(q_k^* + \tilde{q}_k). \tag{6.8}$$

Combined with equations (6.6) and (6.7), this gives the following set of linear equations, which has a tridiagonal coefficient matrix:

$$\begin{gathered} q_{k-1}^*(\tilde{\alpha}_k - \frac{2}{\Delta t} f_k) + q_k^*(\tilde{\beta}_k - \frac{2}{\Delta t} g_k) + q_{k+1}^*(\tilde{\gamma}_k - \frac{2}{\Delta t} h_k) = \\ \tilde{q}_{k-1}(-\tilde{\alpha}_k - \frac{2}{\Delta t} f_k) + \tilde{q}_k(-\tilde{\beta}_k - \frac{2}{\Delta t} g_k) + \tilde{q}_{k-1}(-\tilde{\gamma}_k - \frac{2}{\Delta t} h_k) \end{gathered}. \tag{6.9}$$

Pluvius II's vertical transport-integration subroutines perform the following tasks: Subroutine **`sinteg`** (Table 6.20) performs a general coordination of the vertical integration process. Subroutine **`propss`** (Table 6.21) computes values of A_z, B_z, and C_z for the grid locations z_k at the current model time and stores them in the arrays **`aas, bbs, and ccs,`** respectively.[2] Subroutine **`coefss`** (Table 6.22) uses these arrays to compute the $\tilde{\alpha}_k$, $\tilde{\beta}_k$, and $\tilde{\gamma}_k$, and subsequently the coefficients on the left-hand side of equation (6.9). These values are also stored in **`aas, bbs,`** and **`ccs,`** thus overwriting A_z, B_z, and C_z, which are no longer needed. Subroutine **`bounds`** (Table 6.23) sets appropriate coefficients at the top and bottom extremities of the solution vector. Subroutine **`loads`** (Table 6.24) transfers the appropriate sets o**`f aas, bbs,`** and **`ccs`** to the arrays **`a, b,`** and **`c.`** In addition, it computes the right-hand side of equation (6.9) and stores this in the **`d`** array. Arrays **`a, b, c,`** and **`d`** are used directly by subroutine **`tridag`** to solve the set of linear equations for the values of q_k^* (the numerical approximations to ζ_k at time $t + \Delta t$), which are returned in the array **`v.`**

In contrast to subroutines **`sinteg, propss, coeffs, bounds,`** and **`loads,`** subroutine **`transport`** is application-specific and is intended for modification by the user. The example shown in Table 6.25 calculates rain and snow fall velocities using parameterizations described earlier by Easter and Hales (1984), and also estimates precipitation spread parameters associated with differential settling velocities of different-size hydrometeors. These spread parameters are treated as conventional diffusivities and added to those associated with local air motions.

Numerical integration routines for the x-dimension (**`xinteg, propsx, coefsx, boundx,`** and **`loadx,`** shown in Tables 6.26 through 6.30, respectively) and for the y-dimension (**`yinteg, propsy, coefsy, boundy,`** and **`loady,`** shown in Tables 6.31 through 6.35,

[2]**Note that a separate set of `aas, bbs,`** and **`ccs`** is required for each transport regime.

respectively) proceed through pathways that are similar to those for their vertical-integration counterparts. Because of this they will not be discussed further here, except to note that these routines are somewhat simpler than those for the vertical computations. Fewer physical terms are necessary in these horizontal computations, and they deal directly with the dependent variables **`r(i,j,k,l)`** rather than using their "columized" counterparts.

6.3.3 Numerical Filtering Operations

Filtering of short-wavelength numerical noise, which is generated by the finite-element procedure under conditions where advection dominates as a transport process, occurs through calls to subroutines **`sfilter`**, **`xfilter`**, and **`yfilter`**, (Tables 6.36 through 6.38) which in turn interrogate subroutines **`bndfilt`** and **`foresfilt`**, which are shown in Tables 6.39 and 6.40. Subroutines **`sfilter`**, **`xfilter`**, and **`yfilter`** determine whether filtering is to be applied at the current stage in the computation and, if so, execute the subroutine calls to **`bndfilt`** and **`foresfilt`**, which perform the actual filtering operations for the domain boundaries and internal grid-points, respectively.

As indicated by the listing in Table 6.39, subroutine **`bndfilt`** performs filtering operations only under conditions where boundary inflow occurs, and only when flow conditions exceed specific levels, as determined by the local Peclet number, ($v_i \Delta x_i/K_i$, where the subscript i denotes any appropriate dimension, x, y, or s). Subroutine **`foresfilt`** is based on a subroutine in the Pluvius Mod. 5 code (Easter and Hales 1985), which applies the nonlinear filtering scheme suggested by Forester (1977). The reader is referred to these two references for more extensive discussion on this subject.

6.3.4 Transformation-Integration Operations

Integration of the transformation terms associated with equation (3.2) is coordinated through subroutine **`chmint`**, which is listed in Table 6.41. Because it is more convenient under most circumstances to compute transformation rates in terms of absolute concentrations, the first action of this subroutine is to convert the mixing ratios in the **`rclm`** arrays to concentrations (in moles per cubic centimeter), and store these values in the array **`cnn(l)`**, where **`l`** is the species index. In the special case where the energy balance is calculated and **`l`** corresponds to the equivalent potential temperature, θ_c, these values are converted to actual temperatures (degrees Kelvin) by calling the subroutine **`tfrmth`** (**`cnn(ltheta)`** will contain the actual temperature under these circumstances).

Upon performing these initial operations, **`chmint`** loads a first-guess of the optimum step size for the transformation integration into **`dtsug.`** This value is set to zero if no prior integrations have occurred for that particular grid point, and is determined from the previous integration otherwise. The code subsequently calls subroutine **`odeint,`** which performs the actual numerical integration over the period **`dtchem,`** and then stores the current value of **`dtsug`** in the array **`dtsave`** for possible later use. Upon a successful call to **`odeint, chmint`** restores the concentrations and potential temperature into the **`rclm`** array, and proceeds to the next grid point in the column. After processing an entire column the code returns to the calling subroutine **`core`** for subsequent integration of the transport components.

As noted in Section 3.3, subroutine **`odeint,`** listed in Table 6.42, is only one of several ODE solvers that can be incorporated with Pluvius II to perform integrations of the model's transformation terms, and the user may replace this routine without too much trouble, if desired. **`odeint`** is called by subroutine **`chmint`** giving the grid-point location **`(i,j,k),`** the starting time for the integration **`(tstrto),`** the total integration period for that call **`(dttot),`** the suggested initial step-size **`(dtsug),`** and the maximum allowable step size **`(dtmax)`** as arguments. Additional parameters are passed via common blocks (see commenting at the start of the listing in Table 6.42). Primary output from **`odeint`** is the vector of species concentrations **`cnn(l)`** resulting from the integration process.

This particular **`odeint`** version is a somewhat modernized copy of the code given by Easter and Hales (1984). In essence it is a two-step, low-order, predictor-corrector scheme which has the ability to sense for stiff conditions. Under circumstances involving stiff equations, the scheme shifts to an operation mode where species decay rates are approximated by exponential functions, thus preventing negative undershoot and preserving stability at reasonably large step-sizes.

To implement this, **`odeint`** demands that rate expressions for species generation be segregated from their species-decay counterparts. This is accomplished using the arrays **`ggen(l)`** and **`gdec(l),`** which contain current rates in moles/cm^3 sec and are computed by calls to subroutine **`gen,`** which is a user-supplied subroutine. More detailed information on **`odeint`** is available in the Easter-Hales reference, and the code appearing in Table 6.42 is extensively documented to facillitate its application.

Examples of subroutine **`gen,`** corresponding to Case Example A and Case Example B, are given in Tables 6.43a and 6.43b, respectively. **`gen`** is called by subroutine **`odeint,`** giving the grid-point location **`(i,j,k),`** the current simulation time **`(tgen),`** and the current **`odeint`** step-size **`(dtau)`**. In addition, the argument list contains a flag **`(jpasso)`** to indicate whether the current **`odeint`** call is from the predictor or corrector pass.

Primary output from **`gen`** consists of the vectors of generation and decay rates, as well as species concentrations associated with selected constrained variables. As noted in the context of subroutines **`inptgn`** and **`init,`** Case Example A involves four transported variables: equivalent potential temperature, vapor plus cloud water, rain, and snow. The code in Table 6.43a computes the associated generation and decay rates along the lines of the interaction diagram in Figure 4.1a, using cloud-physics parameterizations described previously by Easter and Hales (1984) and summarized in Table 5.1. The final portion of the subroutine computes corresponding terms of the energy-balance equation, using the derived phase-transformation rates in conjunction with thermodynamic properties, as indicated by the right-hand side of equation (2.9).

The version of subroutine **`gen`** corresponding to Case Example B, shown in Table 6.43b, performs operations that are similar to its Case Example A counterpart, but correspond to the chemical interaction diagrams in Figures 4.1b and 4.1c. Here it should be noted that noted that the transfer of sulfur and nitrogen compounds between physical media depends directly on the cloud-physics processes altering these phases, and so many of the cloud-physics parameters must be computed in this application as well. These operations are performed in the first major segment of the code, which terminates just prior to statement number **`100.`**

Following the cloud-physics computations, the code proceeds to establish current concentrations of the constrained species, and then moves immediately to calculate the **`ggen`** and **`gdec`** arrays for the transported variables. Since the energy-balance component of Pluvius II is inactive for Case Examble B, no latent-heating computations are necessary for this version of **`gen.`** As was the case with Table 6.43a, the calculations exhibited in Table 6.43b are derived mainly from those presented by Easter and Hales (1984) and summarized in Table 5.1, and the reader is referred to these sources for additional information.

6.3.5 Utility Routines for Calculation of Thermodynamic Properties

Various utility subroutines and internal functions for calculating thermodynamic relationships are listed in Tables 6.44 through 6.51. With few exceptions these routines are self-explanatory, and little elaboration will be given here. Subroutine **`henry,`** listed in Table 6.50, is based on SO_2 solubility equations described earlier by Hales and Sutter (1973), and the reader can consult this reference if more information is necessary.

Subroutine **`tfrmth,`** listed in Table 6.51 is used to convert from equivalent potential temperature to actual temperature whenever temperatures are required to evaluate thermodynamic and/or kinetic parameters. Calculations in this subroutine are based on equations (2.7) and (2.8)

$$\theta_c = \theta_d (1 - \frac{\lambda_c r_c}{C_p T}) \quad , \qquad (2.7)$$

$$\theta_d = T(\frac{1000 \text{ mb}}{p})^{R/C_p} \quad , \qquad (2.8)$$

which were discussed previously in Section 2. Because the cloud-water mixing ratio r_c is a constrained variable, its numerical value is generally unknown until T is known. This effectively renders T an implicit function of θ_c, and necessitates an iterative or interpolative technique for calculating temperature. The code in Table 6.51 follows an interpolative approach, using the base data points as described in the code listing.

6.3.6 General Utility Subroutines

Tables 6.52 through 6.56 are listings of miscelaneous utility subroutines. Subroutine **`integrate`,** listed in Table 6.52 is a straightforward Simpson's-rule application, which is used in this code exclusively to integrate fluxes in subroutine **`matbal`** to determine overall inflow and outflow rates. It is nothing fancy, but is relatively efficient and suffices for the purpose at hand. Subroutine **`tridag`,** listed in Table 6.53, is the tridiagonal matrix solver called by subroutines **`sinteg`, `xinegt`,** and **`yinteg`,** and has been taken directly from the textbook of Carnahan, Luther, and Wilkes (1969).

Subroutine **`sbstep`** (Table 6.54) is called from the vertical-integration loop within subroutine **`core`** to determine appropriate time-substeps for the associated integration process. As noted previously, this determination is based on a Courant-number criterion.

The final two subroutines in the code, **`ijpack`** and **`ijunpack`** are shown in Tables 6.55 and 6.56. As noted previously, these routines are executed in conjunction with the vertical-integration step as a method for reducing page-faulting in computers where core memory resources are limited. **`ijpack`** moves key variables from their higher-dimensional arrays into lower-dimensional counterparts representing vertical columns at specified x and y locations. For the dependent variables (in the **`r`** and **`rclm`** arrays), it also moves the updated **`rclm`** values from the last **`sinteg`** and **`chmint`** integrations back into the **`r`** array. **`ijunpack`** performs a clean-up activity, moving updated **`rclm`** values for the final column (**`i = itot, j = jtot`**) back into the **`r`** array.

Table 6.57 provides a summary of the total ensemble of Pluvius subroutines. This is intended primarily for the user's convenience as a locator for the various components of code discussed in this section.

Table 6.1 Pluvius II Code Variable and Array Definitions

Variable Name	**Description**
a	Working array for tridiagonal matrix solution routine.
aas	Working array in vertical finite-element integration scheme.
aax	Working array in x-direction finite-element integration scheme.
aay	Working array in y-direction finite-element integration scheme.
asofhi	Working array in vertical finite-element integration scheme. holds values of **aas** (as defined in **propss**) at lower boundary, **k=1.**
asoflo	Working array in vertical finite-element integration scheme. holds values of **aas** (as defined in **propss**) at upper boundary, **k=ktot.**
axofhi	Similar to **asofhi,** but for **x**-direction.
axoflo	Similar to **asoflo,** but for **x**-direction.
ayofhi	Similar to **asofhi,** but for **y**-direction.
ayoflo	Similar to **asoflo,** but for **y**-direction.
ak	**glc/cp.**
alocp	**heatc/cp.**
b	Working array for tridiagonal matrix solution routine.
bbs	Working array in vertical finite-element integration scheme.
bbx	Working array in x-direction finite-element integration scheme.
bby	Working array in y-direction finite-element integration scheme.
bsofhi	Working array in vertical finite-element integration scheme. holds values of **bbs** (as defined in **propss**) at lower boundary, **k=1.**
bsoflo	Working array in vertical finite-element integration scheme. holds values of **bbs** (as defined in **propss**) at upper boundary, **k=ktot.**
bxofhi	Similar to **bsofhi,** but for **x**-direction.
bxoflo	Similar to **bsoflo,** but for **x**-direction.
byofhi	Similar to **bsofhi,** but for **y**-direction.
byoflo	Similar to **bsoflo,** but for **y**-direction.
c	Working array for tridiagonal matrix solution routine.
cair	Array containing molar air concentrations at grid points.
cairclm	Array holding molar air concentrations for current vertical column of grid points during vertical transport and transformation integrations.
ccs	Working array in vertical finite-element integration scheme.
ccx	Working array in x-direction finite-element integration scheme.
ccy	Working array in y-direction finite-element integration scheme.

Table 6.1, Continued

cerror	Array in ODE integration routine holding concentrations below which the convergence-error criterion is relaxed. Set in subroutine **inptgn.**
cmin	Array in ODE integration routine holding the concentration below which the dependent variable is considered as essentially equal to zero. Set in subroutine **inptgn.**
cnn	Array of molar pollutant concentrations for current grid point during calculations within Pluvius II's transformation subroutines.
csofhi	Working array in vertical finite-element integration scheme. holds values of **ccs** (as defined in **propss**) at lower boundary, **k=1.**
csoflo	Working array in vertical finite-element integration scheme. holds values of **ccs** (as defined in **propss**) at upper boundary, **k=ktot.**
cp	Heat capacity of air at constant pressure.
ct1a - ct · ·	Arrays of coefficients used in the finite-element integration. Computed in subroutine **grdset.**
d	Working array for tridiagonal matrix solution routine.
deltas	Array of vertical grid spacings. Computed in subroutine **grdset.**
deltax	Array of x-grid spacings. Computed in subroutine **grdset.**
deltay	Array of y-grid spacings. Computed in subroutine **grdset.**
diffus	Array of vertical diffusivities at grid points.
diffux	Array of x-direction diffusivities at grid points.
diffuy	Array of y-direction diffusivities at grid points.
diffusclm	Array of vertical diffusivities at grid points in the current vertical column.
drain	Array of effective vertical diffusivities of rain associated with differential fall speeds of different-sized raindrops. (For current vertical column).
dsnow	Array of effective vertical diffusivities of snow associated with differential fall speeds of different snowflakes. (For current vertical column).
dsunif	Vertical-grid spacing for case of uniform vertical grid. If vertical coordinate is inactive or nonuniform, this variable should be set to 0. Set in subroutine **inptgn.**

Table 6.1, Continued

dt	Integration time-step for a x and y transport integration steps. Set in subroutine **inptgn.**
dtcur	Current time-step or time-substep for transport integration.
dtdisk	Control variable for output to backup disk file; backup disk file updated every **dtdisk** seconds after initial disk write, which is controlled by input variable tdisk. Set in subroutine **inptgn.**
dtmin	Minimum allowable time-step size for ODE integrator. Set in subroutine **inptgn.**
dtsave	Array of internal time increments, for each grid-point, associated with the ODE integrator **odeint.** Used for initiating computations on subsequent subroutine calls.
dxunif	Same as for **dsunif**, but for x-grid.
dyunif	Same as for **dsunif** and **dxunif**, but for y-grid.
eps	Error tolerance limit for ODE integration. Set in subroutine **inptgn.**
fdcofs	Array of coefficients for calculating the first derivative of a function or vector of vertical position using second-order finite differencing. Computed in subroutine **grdset.**
fdcofx	Array of coefficients for calculating the first derivative of a function/vector of x using second-order finite differencing. Computed in subroutine **grdset.**
fdcofy	Array of coefficients for calculating the first derivative of a function/vector of y using second-order finite differencing. Computed in subroutine **grdset.**
filfacs	Array holding vertical-integration filter factors for filtering subroutine. Set in subroutine **inptgn.**
filfacx	Array holding x-integration filter factors for filtering subroutine. Set in subroutine **inptgn.**
filfacy	Array holding y-integration filter factors for filtering subroutine. Set in subroutine **inptgn.**
flxhs	Array of vertical pollutant fluxes at **k=ktot** boundary.
flxhsclm	Vertical pollutant fluxes at **k=ktot** boundary associated with the current vertical column of grid points.
flxhx	Array of x-direction pollutant fluxes at **i=itot** boundary.
flxhy	Array of y-direction pollutant fluxes at **j=jtot** boundary.
flxls	Array of vertical pollutant fluxes at **k=1** boundary.
flxlsclm	Vertical pollutant fluxes at **k=1** boundary associated with the current vertical column of grid points.

Table 6.1, Continued

flxlx	Array of vertical pollutant fluxes at **i=1** boundary.
flxly	Array of vertical pollutant fluxes at **j=1** boundary.
gamdry	Dry adiabatic lapse rate.
gdec	Array of species-decay terms computed in subroutine **gen** and applied in the ODE integration process.
ggen	Array of species-generation terms computed in subroutine **gen** and applied in the ODE integration process.
glc	Gas-law constant.
heatc	Latent heat of water condensation.
heatf	Latent heat of water fusion.
heats	Latent heat of sublimation.
ibchis	Array holding toggle for boundary-condition class for each transported variable for the high-z boundary: 1 for flux-specified; 0 for forced outflow. Set in subroutine **inptgn.**
ibchix	Same as for **ibchis**, but for high-x boundary.
ibchiy	Same as for **ibchis,** but for high-y boundary.
ibclos	Same as for **ibchis,** but for low-s boundary.
ibclox	Same as for **ibchis,** but for low-x boundary.
ibcloy	Same as for **ibchis,** but for low-y boundary.
iexp	Toggle in ODE integrator for exponential or Taylor's-series operation: 1 for exponential; 0 for Taylor series. Set in subroutine **inptgn.**
ijkmax	Maximum of **imaxd, jmaxd and kmaxd.**
imaxd	Dimensioning limit for x grid points in simulation.
initmode	Initialization control variable; if 1, then subroutine **restart** and historical disk data used in initialization process; otherwise subroutine **restar**t not interrogated. Set in subroutine **inptgn.**
iprint	Counter for output to file **diskoutpt.** Results written to disk file if **iprint** is greater or equal to **nprint.**
itdisk	Disk output control variable for backup disk; if 1, then archive full historical record; if 2, then maintain a single frame, which is updated in time. Set in subroutine inptgn.
itot	Number of x grid points in simulation. Set in subroutine **inptgn.**
jfog	Array holding flags indicating the presence of local cloud water: 1 if cloud water present; 0 otherwise.
jmaxd	Dimensioning limit for y grid points in simulation.
jtot	Maximum number of y grid points in simulation. Set in subroutine **inptgn.**

Table 6.1, Continued

kmaxd	Dimensioning limit for z grid points in simulation.
ktot	Maximum number of z grid points in simulation. Set in subroutine **inptgn.**
lcloud	Species index for cloud water.
ldifusmx	Number of diffusion regimes in simulation.
lhcloud	Species index for hydrogen ion in cloud water
lhrain	Species index for hydrogen ion in rain.
lh2o2gcl	Species index for gaseous + cloud-water hydrogen peroxide
lh2o2rain	Species index for rainborne hydrogen peroxide.
lh2o2snow	Species index for snowborne hydrogen peroxide
lmaxd	Dimensioning limit for transported variables in simulation.
lno3g	Species index for gaseous nitric acid.
lno3gcl	Species index for gaseous nitric acid + cloud-water nitrate ion.
lno3rain	Species index for rainborne nitrate ion.
lno3snow	Species index for snowborne nitrate ion.
lozone	Species index for gas-phase ozone.
lprops	Array holding the z-direction transport-regime index for each transported species. Set in subroutine **inptgn.**
lpropx	Array holding the x-direction transport-regime index for each transported species. Set in subroutine **inptgn.**
lpropy	Array holding the y-direction transport-regime index for each transported species. Set in subroutine **inptgn.**
lpsmax	Dimensioning limit for number of z transport regimes in simulation.
lpstot	Number of z transport regimes in simulation. Set in subroutine **inptgn.**
lpxmax	Dimensioning limit for number of x transport regimes in simulation.
lpxtot	Number of x transport regimes in simulation. Set in subroutine **inptgn.**
lpymax	Dimensioning limit for number of y transport regimes in simulation.
lpytot	Number of y transport regimes in simulation. Set in subroutine **inptgn.**
lrain	Species index for rain.
lsnow	Species index for snow.
lso2gcl	Species index for gaseous SO_2 + S(IV) in cloud water.
lso2rain	Species index for S(IV) in rain.
lso2snow	Species index for S(IV) in snow.

Table 6.1, Continued

lso4g	Species index for sulfate aerosol.
lso4gcl	Species index for sulfate aerosol + sulfate ion in cloud water.
lso4rain	Species index for sulfate ion in rain.
lso4snow	Species index for sulfate ion in snow.
ltemp	Index for temperature.
ltheta	Index for equivalent potential temperature.
ltot	Number of transported species in simulation. Set in subroutine **inptgn.**
ltot2	Number of transported plus constrained species in simulation. Set in subroutine **inptgn.**
lvapcl	Species index for water vapor + cloud water
l2maxd	Dimensioning limit for transported plus constrained species. **l2maxd = lmaxd + mmaxd.**
mbndfils	Toggle controlling additional filtering at upper and lower inflow boundaries. If positive, the additional filtering is applied at the respective coordinate boundaries.
mbndfilx	Similar to mbndfils, but for x- boundaries.
mbndfily	Similar to mbndfils, but for y- boundaries.
mfreqps	Frequency at which transport-property variables are updated (through subroutine **propss**) during an z-integration step. Set in subroutine **inptgn.** legal values are: 1 - called once each time in **core,** during the first **zinteg** call 10, 20 - consider z-integrations as paired within each subintegration step (**nsubt** pairs) 10 - called once during first of the pair 20 - called once during each of the pair (if **nsubt**=1, then the 10 and 20 options are like the 1 and 2 options for x and y). Set in subroutine **inptgn.**
mfreqpx	Similar to **mfreqps,** but pertains to x integration legal values are: 0 - called only once at start of program - transport properties are time invariant 1 - called once during the first of the two x-integration in **core** 2 - called once during each of the two x-integrations in **core** Set in subroutine **inptgn.**
mfreqpy	Similar to **mfreqp**x, but pertains to y-integration.

Table 6.1, Continued

mgen	Control variable for chemical/physical transformation integration; if 1, then the transformation integrations are performed; if 0, they are skipped. Set in subroutine **inptgn.**
mmaxd	Dimensioning limit for constrained species in simulation.
mtemp	Energy balance integration control variable: 1 - energy balance calculated and equivalent potential temperature is a transported variable; 0 - energy balance is not calculated: temperature is a constrained variable. Set in subroutine **inptgn.**
mtitle	Character variable containing title of simulation.
mtrans	Transport integration control variable; if 1, then z integrations are performed; if 0, then z integrations are skipped. Set in subroutine **inptgn.**
mtranx	Transport integration control variable; if 1, then x integrations are performed; if 0, then x integrations are skipped. Set in subroutine **inptgn.**
mtrany	Transport integration control variable; if 1, then y integrations are performed; if 0, then y integrations are skipped. Set in subroutine **inptgn.**
munifs	Specifies uniform/non-uniform grid in vertical direction: 1 if uniform; 0 otherwise. Set in subroutine **inptgn.**
munifx	Specifies uniform/non-uniform grid in x-direction: 1 if uniform; 0 otherwise. Set in subroutine **inptgn.**
munify	Specifies uniform/non-uniform grid in y-direction: 1 if uniform; 0 otherwise. Set in subroutine **inptgn.**
name	Character arrays containing names of species in simulation.
ncorr	Number of corrector passes applied in ODE integrator. Set in subroutine **inptgn.**
nfilsrch	Parameter in filter subroutine (see subroutine **foresfilt** listing in Table 40). Set in subroutine **inptgn.**
nfiltitr	Parameter in filter subroutine (see subroutine **foresfilt** listing in Table 40). Set in subroutine **inptgn.**
nfilts	Number of z-integration steps between filtering. Set in subroutine **inptgn.**
nfiltset	Parameter in filter subroutine (see subroutine **foresfilt** listing in Table 40). Set in subroutine **inptgn.**
nfiltx	Number of x-integration steps between filtering. Set in subroutine **inptgn.**

Table 6.1, Continued

nfilty	Number of y-integration steps between filtering. Set in subroutine **inptgn.**
nnammx	Dimensioning limit for name array: **nnammx = lmaxd+mmaxd plus 1** extra for **cair**.
nonsym	Toggle for nonsymmetric integration in subroutine **core.** If positive, integration is nonsymmetric; otherwise, full symmetric integration is performed.
nprint	Control variable for output to file **diskoutpt.** Output is written after **nprint** composite integration steps by subroutine **core.** Set in subroutine **inptgn.**
ptot	Array of atmospheric pressures at grid points.
ptotclm	Array holding atmospheric pressures at grid points in the current vertical column.
r	Array holding pollutant-species mixing ratios or temperatures.
ratesf	Array in ODE integrator holding the decay rate, for each dependent variable, at which the associated ODE is considered stiff. Set in subroutine inptgn.
rclm	Array holding pollutant-species mixing ratios in a single vertical column.
scord	Array holding locations of vertical grid points. Set in subroutine **grdxyz**.
t	Current simulation time. Inside of subroutine **core, t** is the start-time of the current integration substep; outside of subroutine **core, t** is the current model time. Initialized in subroutine **inptgn**.
tdisk	Control variable for output to backup diskfile by subroutine **collect**; first output occurs when simulation reaches or exceeds **tdisk.**
trestart	Control variable for input from previous backup disk file by subroutine **restart,** when **initmode = 1.** If positive, the time-frame having a simulation time equal to **trestart** is used; if negative, the first time frame on the file is used.
trun	Simulation time at end of current integration substep.
tstop	Simulation time at which code execution is to terminate; set in subroutine **inptgn**.
v	Working array for tridiagonal matrix solution routine.
vertclm	Array of vertical fall velocities of media (rain and snow in Case Examples A and B), in the current vertical column of grid points.

Table 6.1, Continued

winds	Array of vertical wind components at grid points. Computed in subroutine **wind.**
windsclm	Array of vertical wind components in the current vertical column of grid points.
windx	Array of x wind components at grid points. Computed in subroutine **wind.**
windy	Array of y wind components at grid points. Computed in subroutine **wind.**
xcord	Array holding locations of x grid points. Set in subroutine **grdxyz**
ycord	Array holding locations of y grid points. Set in subroutine **grdxyz.**

Table 6.2: Listing of Main Program **pluvius**

```
      program pluvius
      include "pluvius.com"
c
c      Main program for pluvius code:  initializes, then cycles through
c      progressive time steps and terminates when model time equals tstop.
c
c     Enter control variables, various initialization parameters, and
c       computation grid.
c
      call inptgn
c
c     Set up finite-element parameters.
c
      call femset
c
c     Set initial values of dependent-variable arrays.
c
      call init
c
c     Input initial wind field.
c
      call wind
c
c     Set up output files and write initial dependent-variable arrays.
```

Table 6.2, Continued

```
c
       call printi
c
c      Compute inflow and outflow fluxes and amounts.
c
       call matbal
c
c      Initialize print counter.
c
       iprint = 0
c
c      Start major computation loop:  compute wind field at time t.
c
10     call wind
c
c      Compute diffusivity field at time t.
c
       call diff
c
c      Integrate over time step.
c
       call core
c
c      Clean arrays of fragmentary values.
c
       call cleanup
c
c      Store data on binary file for possible restart of code.
c
       if(itdisk .ne. 0.and.t.gt.tdisk-.001*dt) then
         call collect
       end if
       iprint = iprint+1
       if (iprint.ge.nprint) then
c
c      Output results to disk.
c
         call print
c
c      Compute inflow and outflow fluxes and amounts.
c
         call matbal
         iprint = 0
```

Table 6.2, Continued

```
       endif
c
c       The following is a "security blanket" to inform the user that
c       computation is proceeding. It prints to screen the current time,
c       plus three parameters at grid point 1,19,15 in the interior of the
c       computational domain:
c         1. the temperature
c         2. the saturation mixing ratio of water vapor
c         3. the sum of cloud+vapor water minus the saturation value
c       This print block should be removed as soon as the user has
c       sufficient confidence that the test codes in Examples A and B
c       are operating properly.
c
       rr=sat(r(1,19,15,ltemp))/cair(1,19,15)
        dif= r(1,19,15,lvapcl)-rr
        write(6,21) t,r(1,19,15,ltemp),rr,dif
21    format(1h ,'time, temp, sat, diff = ',4(e12.6,1x))
c
      if (t.lt.tstop) go to 1O
       stop
       end
```

Table 6.3a: Listing of Include File **`pluvius.com`** for Case Example A

```
c-------------------------------------------------------------------
c
c       pluvius II include file for Case Example A -- met. only
c
c-------------------------------------------------------------------------
        parameter( imaxd=1, jmaxd=33, kmaxd=37)
        parameter( ijkmax = 37)
        parameter( lmaxd=4, l2maxd=7, mmaxd=3 )
        parameter( lpxmax=1, lpymax=2, lpsmax=4 )
        parameter( ldifusmx=lpsmax )
        parameter( nnammx= l2maxd+mmaxd+2 )
c
```

Table 6.3a, Continued

```
c       imaxd, jmaxd, kmaxd, = dimensioning limits for x, y, and z.
c       ijkmax = maximum of imaxd,jmaxd, and kmaxd.
c       lmaxd = dimensioning limit for transported species.
c       l2maxd = dimensioning limit for species contained in r array.
c       mmaxd = dimensioning for steady-state species.
c       lpxmax,lpymax,lpsmax= dimensioning limits for consolidated parameters:
c         normally lpxmax=lpymax=1 and lpsmax may be 3 or 4.
c       ldifusmx = dimensioning limit for vertical eddy diffusivity.
c         ldifusmx = 1 when all vertical regimes have same diffusivities.
c         ldifusmx = lpsmx when regimes have different diffusivities.
c
c       Set key thermodynamic parameters.
c
        parameter( gamdry =-9.76e-5)  !dry adiabatic lapse rate (deg. k/cm)
        parameter( cp=2.91e8)    !air heat capacity  (erg/mole deg.k)
        parameter( glc=8.314e7)     !gas-law const.  (erg/mole deg. k)
        parameter( heatc=4.5e11)     !lat. heat of cond. (erg/mole)
        parameter( heats=5.02e11)   !lat. heat of sub.  (erg/mole)
        parameter( heatf=5.2e10)    !lat. heat of fusion (erg/mole)
        parameter( ak=0.286)        !r/cp
        parameter( alocp=1725.)     !heatc/cp (deg. k)
c
c       Number of grid points, species, transport regimes.
c
        common /arrays/itot,jtot,ktot,ltot,ltot2,lpstot,
     +  lpytot,lpxtot
c
c       itot, jtot, ktot = number of x, y, and z grid points.
c       ltot = actual number of transported species.
c       ltot2 = actual number of species in r array.
c       lpstot = actual number of phases used to compute vertical transport.
c       lpytot, lpxtot = ditto for y- and x-directions.
c
c       Species names and output title.
c
        common /names/ name, mtitle
        character mtitle*(80)
        character*10 name(nnammx)
c
c       Note:  these names are for printout only, and control nothing else
c       within the code.
c
c       name = 10 character species name
```

Table 6.3a, Continued

```
c       mtitle = 8O character run title
c
c       Index location names for =dependent variables.
c
        common /locns/ lvapcl,lrain,lsnow,lcloud,ltemp,lwvap,
     +  ltheta
c
c       Arrays for tridiagonal matrix routine.
c
        common /trdag/ a(ijkmax), b(ijkmax), c(ijkmax), d(ijkmax),
     +    v(ijkmax)
c
c       These arrays are used to hold tri-diagonal matrix coefficients and
c       solution.  tridag solves system of linear equations:
c       a(i-1)*v(i-1) + b(i)*v(i) + c(i+1)*v(i+1) = d(i)
c
        common /regime/ lpropx(l2maxd), lpropy(l2maxd), lprops(l2maxd)
c
c       Holds the x-regime, y-regime, and z-regime for each species.  In
c       z-direction, species are grouped by regime. Typically will have
c       rain-borne species, snow-borne species, and others (with negligible
c       fall speeds).  For x and y, will likely be just one regime.
c
c       Timing variables and controls.
c
        common /temprl/ t, trun, dt, tstop, dtcur
c
c       t = model time.  During an x-y-z-chem-z-y-x integration, t is the time
c        of the start of the current integration sub-step, and trun is the
c        time of the end of the integration sub-step.  outside of core, t
c        holds the current time.
c       trun - See above.
c       dt = integration time-step for single x and y integration.
c       tstop = simulation end time.
c       dtcur = current integration time step for x and y, dtcur = dt. For z,
c       dtcur = dtsubt, since z integration may be broken into further time
c        steps.
c
        common /cntrl/ mfreqpx, mfreqpy, mfreqps, nonsym, mgen,
     +  munifx, munify, munifs, mtemp, mtranx, mtrany, mtrans
c
c       mfreqpx = frequency at which propsx is called during x integrations.
```

Table 6.3a, Continued

```
c       legal values are:
c        1 - called once during the first of the 2 x-integration in core.
c        2 - called once during each of the 2 x-integrations in core.
c       mfreqpy - similar but for propsy.
c       mfreqpz - similar but for propss.
c        O - called once at start of program.
c        1 - called once each time in core, during the first zinteg call.
c        1O, 2O - consider z-integrations as paired within each
c         subintegration step (nsubt pairs).
c        1O - called once during first of the pair.
c        2O - called once during each of the pair.
c         (if nsubt=1, then the 1O and 2O options are like the 1 and 2
c         options for x and y.)
c       nonsym - if positive, a non-symmetric integration is performed - just
c        x-y-z-chem.
c       mgen - subroutine chmint is only called if mgen>O.
c       munifsx, munify, munifs - a positive value means x, y, or s/z grid
c        spacing is uniform.  used in filtering routine to improve efficiency.
c       mtranx, mtrany, mtrans - if positive, then x,y, or s transport is
c        computed.  if negative or zero, transport is skipped.
c       mtemp = control variable for energy-balance calculations.
c        1 - energy balance calculated: temperature is a dependent variable
c        O - energy balance is not calculated:  temperature is a constrained
c        variable (if used at all).
c
c       Grid structures and associated computational factors in
c       finite-element determination.
c
        common /grid/
     +    ct1a(3,ijkmax), ct1c(3,ijkmax), ct3b(3,ijkmax),
     +    ct3h(3,ijkmax), ct3a(3,ijkmax), ct3d(3,ijkmax),
     +    ct3f(3,ijkmax), ct3c(3,ijkmax), ct4b(3,ijkmax),
     +    ct4d(3,ijkmax), ct3e(3,ijkmax), ct3df(3,ijkmax),
     +    ct2a, ct2c,
     +    xcord(imaxd), deltax(imaxd), fdcofx(3,imaxd),
     +    ycord(jmaxd), deltay(jmaxd), fdcofy(3,jmaxd),
     +    scord(kmaxd), deltas(kmaxd), fdcofs(3,kmaxd),
     +    xwidth(imaxd),ywidth(jmaxd),swidth(kmaxd)
c
c       xcord(i) = x coordinate at grid point i
c       ycord(i) = y coordinate at grid point j
c       scord(i) = s coordinate at grid point k
c
```

Table 6.3a, Continued

```
c       deltax(i) = xcord(i+1) - xcord(i)
c       deltay(j) = ycord(j+1) - ycord(j)
c       deltas(k) = scord(k+1) - scord(k)
c
c       fdcofx(.,i) = coefficients for calcultating the first derivative of a
c        function/vector using normal finite differencing:
c        dq/dx(i) = fdcofx(1,I)*q(I-1) + fdcofx(2,i)*q(i) + fdcofx(3,i)*q(i+1)
c        ** The coefficients are not defined at i=1 and i=itot. **
c       fdcofy, fdcofs - same idea
c
c       ct...  = coefficients used in the f.e.m. solutions of the transport
c        equations.   The first index refers to x (1), y (2) or s/z (3).  The
c        second refers to grid point.
c
c       Dependent variables (mixing ratio for chemistry and deg. k
c       for temperature).
c
        common /depen/ r(imaxd,jmaxd,kmaxd,l2maxd)
c
c       r(i,j,k,l) = mixing ratio of species l at grid i,j,k.
c
        common /air/ cair(imaxd,jmaxd,kmaxd),ptot(imaxd,jmaxd,kmaxd)
c
c       cair = air density at i,j,k.  units = moles/cc
c       ptot = air pressure at i,j,k.  units = dynes/cm**2
c
c       Arrays for x-transport.
c
        common /xcomon/
     +    windx(imaxd,jmaxd,kmaxd),      diffux(imaxd,jmaxd,kmaxd),
     +    aax(imaxd,jmaxd,kmaxd,lpxmax), bbx(imaxd,jmaxd,kmaxd,lpxmax),
     +    ccx(imaxd,jmaxd,kmaxd,lpxmax),
     +    axoflo(jmaxd,kmaxd,lpxmax),    bxoflo(jmaxd,kmaxd,lpxmax),
     +    axofhi(jmaxd,kmaxd,lpxmax),    bxofhi(jmaxd,kmaxd,lpxmax),
     +    flxlx(jmaxd,kmaxd,lmaxd),      flxhx(jmaxd,kmaxd,lmaxd),
     +    ibchix(lpxmax), ibclox(lpxmax)
c
c       windx = wind-speed in x direction
c       diffux = eddy diffusivity in x direction
c
c       aax, bbx, ccx = arrays used during transport computations.
c        These arrays have different meaning in different parts of the code.
```

Table 6.3a, Continued

```
c        In propsx, we define:
c           dr/dt = aax * d2r/dx2  +  bbx * dr/dx  +  ccx * r
c           (Note:  for x, ccx is zero.  for s/z, ccs is non-zero.)
c        These values are used by coeffs to define new aax, bbx, ccx with
c        meaning
c           aax(i,..)*r(i-1,..) + bbx(i,..)*r(i,..)
c                 + ccx(i,..)*r(i+1,..) + oldint(i,..) = O.
c
c       axoflo,bxoflo-hold values of aax & bbx (as defined in propsx) at i=1.
c       axofhi,bxofhi-hold values of aax & bbx (as defined in propsx) at
c        i=itot.
c       flxlx, flxhs - hold fluxes at i=1 and i=itot boundaries.
c       ibchix, ibclox - boundary condition specifier.  Note that these are
c        defined for each regime, not each species.
c        1 = flux b.c. if inflow or no-flow at a boundary, but outflow b.c. if
c        there is outflow.
c        Anything else = "forced outflow" which gives zero flux if there
c        turns out to be inflow.
c
c       Arrays for y-transport.
c
        common /ycomon/
     +    windy(imaxd,jmaxd,kmaxd),      diffuy(imaxd,jmaxd,kmaxd),
     +    aay(imaxd,jmaxd,kmaxd,lpymax), bby(imaxd,jmaxd,kmaxd,lpymax),
     +    ccy(imaxd,jmaxd,kmaxd,lpymax),
     +    ayoflo(imaxd,kmaxd,lpymax),    byoflo(imaxd,kmaxd,lpymax),
     +    ayofhi(imaxd,kmaxd,lpymax),    byofhi(imaxd,kmaxd,lpymax),
     +    flxly(imaxd,kmaxd,lmaxd),      flxhy(imaxd,kmaxd,lmaxd),
     +    ibchiy(lpymax), ibcloy(lpymax)
C
c       Arrays for vertical transport.
c
        common /scomon/
     +    winds(imaxd,jmaxd,kmaxd),diffus(imaxd,jmaxd,kmaxd,lpsmax),
     +    vertcl(imaxd,jmaxd,kmaxd,lpsmax),
     +    aas(kmaxd,lpsmax),    bbs(kmaxd,lpsmax),   ccs(kmaxd,lpsmax),
     +    asoflo(lpsmax),       bsoflo(lpsmax),      csoflo(lpsmax),
     +    asofhi(lpsmax),       bsofhi(lpsmax),      csofhi(lpsmax),
     +    flxls(imaxd,jmaxd,lmaxd), flxhs(imaxd,jmaxd,lmaxd),
     +    ibchis(lpsmax), ibclos(lpsmax),
     +    jfog(imaxd,jmaxd,kmaxd), dbydz2(ijkmax),
     +    drain(kmaxd), dsnow(kmaxd), vrain(kmaxd), vsnow(kmaxd)
c
```

Table 6.3a, Continued

```
c       Much of this parallels xcomon.  However, since s/z integrations are
c       performed one column at a time, most of the computational variables
c       have no i-j dimensions.
c
c       diffus = vertical eddy diffusivity.  Note fourth dimension which
c        may be used if one does energy equation and has different diffusivity
c        for energy and trace substances.
c       vertcl = settling velocity for each regime.  Negative (or zero) except
c        for helium.
c       jfog=saturation flag- jfog = 1 if air is saturated or supersaturated.
c       drain, dsnow = pseudo-diffusivity for rain and snow.
c       vrain, vsnow = vertical fall velocities for rain and snow.
        common /column/ rclm(kmaxd,l2maxd), windsclm(kmaxd),
     +    cairclm(kmaxd), difusclm(kmaxd,lpsmax),ptotclm(kmaxd),
     +    vertcclm(kmaxd,lpsmax), flxlsclm(lmaxd), flxhsclm(lmaxd)
c
c       z-chem-z integrations are performed one "column" at a time. During the
c       integration, variables are moved from their 3 or 4 dimensional arrays
c       into these smaller "_clm" arrays, then back at the integration end.
c       this reduces page faulting.
c
        common / print_com /  nprint, dtdisk, tdisk, itdisk
c
c       nprint = subroutine print is called from main program after
c       nprint calls to core
c       dtdisk = disk output at increments of dtdisk
c       tdisk = next disk output at this time
c       itdisk = O - no disk output
c            other - disk output occurs
c       Note - tdisk is updated as output occurs.  it can be
c       initialized to zero (get initial output) or non-zero values.
c
        common / filter / nfiltx, nfilty, nfilts,
     +     nfilsrch,nfiltset,nfiltitr,
     +     filfacx(lmaxd), filfacy(lmaxd), filfacs(lmaxd),
     +     mbndfilx, mbndfily, mbndfils
c
c       nfiltx = filtering in x direction will occur after this many
c        x-integrations.
c       filfacx(l) = x-direction filter factor for species l.  if filtering
c        is to be done, use O.1O.  for no filtering, use O.
c        (this may be unnecessary.  normally one would filter all or no
```

Table 6.3a, Continued

```
c        species in the x-direction.)
c       nfilty, filfacy - same but for y.
c       nfilts, filfacs - same but for s/z.  in this case, one might wish to
c        filter only the precipitation bound species.
c       mbndfilx, mbndfily, mbndfils - controls the additional filtering at
c        inflow boundaries.  if the value is positive, the additional
c        filtering applies at the respective coordinate boundary.
c
c
c       odeint stuff: see odeint routine for definitions.
c
        common cmin(lmaxd),ratesf(lmaxd), cerror(lmaxd),
     +  cnn(l2maxd),dtsave(imaxd,jmaxd,kmaxd),ggen(lmaxd),gdec(lmaxd),
     +  eps,dtmin,ncorr,iexp
c
        common / inflowcmn /
     +      rinflowlox(jmaxd,kmaxd,lmaxd),
     +      rinflowhix(jmaxd,kmaxd,lmaxd),
     +      rinflowloy(imaxd,kmaxd,lmaxd),
     +      rinflowhiy(imaxd,kmaxd,lmaxd),
     +      rinflowlos(imaxd,jmaxd,lmaxd),
     +      rinflowhis(imaxd,jmaxd,lmaxd)
c
c       rinflowlox(j,k,l) = inflow mixing ratio of species l at grid 1,j,k
c       rinflowhix(j,k,l) = inflow mixing ratio of species l at grid itot,j,k
c       rinflowloy(i,k,l) = inflow mixing ratio of species l at grid i,1,k
c       rinflowhiy(i,k,l) = inflow mixing ratio of species l at grid i,jtot,k
c       rinflowlos(i,j,l) = inflow mixing ratio of species l at grid i,j,1
c       rinflowhis(i,j,l) = inflow mixing ratio of species l at grid i,j,ktot
c
c       rinflowlos would only be used if the lower s/z boundary were not the
c       earth's surface and inflow across this boundary were possible. For
c       theta-c, rinflow--- is the inflow theta-c value.
```

Table 6.3b: Listing of Include File **pluvius.com** for Case Example B

```
c-----------------------------------------------------------------
c
c      pluvius II include file for Case Example B -- chem. only
c
c-----------------------------------------------------------------------
       parameter( imaxd=1, jmaxd=33, kmaxd=37)
       parameter( ijkmax = 37)
       parameter( lmaxd=15, l2maxd=22, mmaxd=7 )
       parameter( lpxmax=1, lpymax=2, lpsmax=4 )
       parameter( ldifusmx=lpsmax )
       parameter( nnammx= l2maxd+mmaxd+2 )
c
c       imaxd, jmaxd, kmaxd, = dimensioning limits for x, y, and z.
c       ijkmax = maximum of imaxd,jmaxd, and kmaxd.
c       lmaxd = dimensioning limit for transported species.
c       l2maxd = dimensioning limit for species contained in r array.
c       mmaxd = dimensioning for steady-state species.
c       lpxmax,lpymax,lpsmax=dimensioning limits for consolidated parameters:
c         normally lpxmax=lpymax=1 and lpsmax may be 3 or 4.
c       ldifusmx = dimensioning limit for vertical eddy diffusivity.
c         ldifusmx = 1 when all vertical regimes have same diffusivities.
c         ldifusmx = lpsmx when regimes have different diffusivities.
c
c       Set key thermodynamic parameters.
c
        parameter( gamdry =-9.76e-5)  !dry adiabatic lapse rate (deg. k/cm)
        parameter( cp=2.91e8)    !air heat capacity  (erg/mole deg.k)
        parameter( glc=8.314e7)     !gas-law const.  (erg/mole deg. k)
        parameter( heatc=4.5e11)     !lat. heat of cond. (erg/mole)
        parameter( heats=5.02e11)   !lat. heat of sub.  (erg/mole)
        parameter( heatf=5.2e10)    !lat. heat of fusion (erg/mole)
        parameter( ak=0.286)        !r/cp
        parameter( alocp=1725.)     !heatc/cp (deg. k)
c
c      Number of grid points, species, and transport regimes.
c
       common /arrays/itot,jtot,ktot,ltot,ltot2,lpstot,
     + lpytot, lpxtot
c
c      itot, jtot, ktot = number of x, y, and z grid points
c      ltot = actual number of transported species
c      ltot2 = actual number of species in r array
```

Table 6.3b, Continued

```
c      lpstot = actual number of phases used to compute vertical transport
c      lpytot, lpxtot = ditto for y- and x-directions
c
c      Species names and output title.
c
       common /names/ name, mtitle
       character mtitle*(80)
       character*10 name(nnammx)
c
c       Note:  these names are for printout only, and control nothing else
c       within the code.
c
c      name = 10 character species name
c      mtitle = 80 character run title
c
c      Location names for dependent variables.
c
      common /locns/ lvapcl,lrain,lsnow,lcloud,ltemp,lhrain,
     +  lhcloud,lso2gcl,lso2rain,lso2snow,lso4g,lso4gcl,
     +  lso4rain,lso4snow,lno3g,lno3gcl,lno3rain,lno3snow,
     + lh2o2gcl,lh2o2rain,lh2o2snow,lozone
c
c       Arrays for tridiagonal matrix routine.
c
        common /trdag/ a(ijkmax), b(ijkmax), c(ijkmax), d(ijkmax),
     +    v(ijkmax)
c
c       These arrays are used to hold tri-diagonal matrix coefficients and
c       solution.  tridag solves system of linear equations:
c       a(i-1)*v(i-1) + b(i)*v(i) + c(i+1)*v(i+1) = d(i)
c
      common /regime/ lpropx(l2maxd), lpropy(l2maxd), lprops(l2maxd)
c
c       Holds the x-regime, y-regime, and z-regime for each species.  In
c       z-direction, species are grouped by regime. Typically will have
c       rain-borne species, snow-borne species, and others (with negligible
c       fall speeds).  For x and y, will likely be just one regime.
c
c      Timing variables and controls.
c
       common /temprl/ t, trun, dt, tstop, dtcur
c
c       t = model time.  During an x-y-z-chem-z-y-x integration, t is the time
```

Table 6.3b, Continued

```
c        of the start of the current integration sub-step, and trun is the
c        time of the end of the integration sub-step.  outside of core, t
c        holds the current time.
c       trun - See above.
c       dt = integration time-step for single x and y integration.
c       tstop = simulation end time.
c       dtcur = current integration time step for x and y, dtcur = dt. For z,
c       dtcur = dtsubt, since z integration may be broken into further time
c        steps.
c
      common /cntrl/ mfreqpx, mfreqpy, mfreqps, nonsym, mgen,
     + munifx, munify, munifs, mtemp, mtranx, mtrany, mtrans
c
c       mfreqpx = frequency at which propsx is called during x integrations.
c       legal values are:
c        0 - called only once at start of program - transport properties
c         are time-invariant.
c        1 - called once during the first of the 2 x-integration in core.
c        2 - called once during each of the 2 x-integrations in core.
c       mfreqpy - similar but for propsy.
c       mfreqpz - similar but for propss.
c        1 - called once each time in core, during the first zinteg call.
c        10, 20 - consider z-integrations as paired within each
c           subintegration step (nsubt pairs).
c        10 - called once during first of the pair.
c        20 - called once during each of the pair.
c         (if nsubt=1, then the 10 and 20 options are like the 1 and 2
c         options for x and y.)
c       nonsym - if positive, a non-symmetric integration is performed - just
c        x-y-z-chem.
c       mgen - subroutine chmint is only called if mgen>0.
c       munifsx, munify, munifs - a positive value means x, y, or s/z grid
c        spacing is uniform.  used in filtering routine to improve efficiency.
c       mtranx, mtrany, mtrans - if positive, then x,y, or s transport is
c        computed.  if negative or zero, transport is skipped.
c       mtemp = control variable for energy-balance calculations.
c        1 - energy balance calculated: temperature is a dependent variable
c        0 - energy balance is not calculated:  temperature is a constrained
c        variable (if used at all).
```

Table 6.3b, Continued

```
c       Grid structures and associated computational factors in
c       finite-element determination.
c
       common /grid/
     +    ct1a(3,ijkmax), ct1c(3,ijkmax), ct3b(3,ijkmax),
     +    ct3h(3,ijkmax), ct3a(3,ijkmax), ct3d(3,ijkmax),
     +    ct3f(3,ijkmax), ct3c(3,ijkmax), ct4b(3,ijkmax),
     +    ct4d(3,ijkmax), ct3e(3,ijkmax), ct3df(3,ijkmax),
     +    ct2a, ct2c,
     +    xcord(imaxd), deltax(imaxd), fdcofx(3,imaxd),
     +    ycord(jmaxd), deltay(jmaxd), fdcofy(3,jmaxd),
     +    scord(kmaxd), deltas(kmaxd), fdcofs(3,kmaxd),
     +    xwidth(imaxd),ywidth(jmaxd),swidth(kmaxd)
c
c       xcord(i) = x coordinate at grid point i
c       ycord(i) = y coordinate at grid point j
c       scord(i) = s coordinate at grid point k
c
c       deltax(i) = xcord(i+1) - xcord(i)
c       deltay(j) = ycord(j+1) - ycord(j)
c       deltas(k) = scord(k+1) - scord(k)
c
c       fdcofx(.,i) = coefficients for calcultating the first derivative of a
c        function/vector using normal finite differencing:
c        dq/dx(i) = fdcofx(1,i)*q(i-1) + fdcofx(2,i)*q(i) + fdcofx(3,i)*q(i+1)
c        ** The coefficients are not defined at i=1 and i=itot. **
c       fdcofy, fdcofs - same idea
c
c       ct...  = coefficients used in the f.e.m. solutions of the transport
c        equations.   The first index refers to x (1), y (2) or s/z (3).  The
c        second refers to grid point.
c
c       Dependent variables (mixing ratio for chemistry and deg. k
c       for temperature).
c
        common /depen/ r(imaxd,jmaxd,kmaxd,l2maxd)
c
c       r(i,j,k,l) = mixing ratio of species l at grid i,j,k.
c
        common /air/ cair(imaxd,jmaxd,kmaxd),ptot(imaxd,jmaxd,kmaxd)
c
c       cair = air density at i,j,k.  units = moles/cc
```

Table 6.3b, Continued

```
c       ptot = air pressure at i,j,k.  units = dynes/cm**2
c
c       Arrays for x-transport.
c
       common /xcomon/
     +    windx(imaxd,jmaxd,kmaxd),      diffux(imaxd,jmaxd,kmaxd),
     +    aax(imaxd,jmaxd,kmaxd,lpxmax), bbx(imaxd,jmaxd,kmaxd,lpxmax),
     +    ccx(imaxd,jmaxd,kmaxd,lpxmax),
     +    axoflo(jmaxd,kmaxd,lpxmax),    bxoflo(jmaxd,kmaxd,lpxmax),
     +    axofhi(jmaxd,kmaxd,lpxmax),    bxofhi(jmaxd,kmaxd,lpxmax),
     +    flxlx(jmaxd,kmaxd,lmaxd),      flxhx(jmaxd,kmaxd,lmaxd),
     +    ibchix(lmaxd), ibclox(lmaxd)
c
c       windx = wind-speed in x direction
c       diffux = eddy diffusivity in x direction
c
c       aax, bbx, ccx=computational arrays used during transport computations.
c        These arrays have different meaning in different parts of the code.
c        In propsx, we define:
c           dr/dt = aax * d2r/dx2  +  bbx * dr/dx  +  ccx * r
c           (Note:  for x, ccx is zero.  for s/z, ccs is non-zero.)
c        These values are used by coeffs to define new aax, bbx, ccx with
c        meaning
c           aax(i,..)*r(i-1,..) + bbx(i,..)*r(i,..)
c                 + ccx(i,..)*r(i+1,..) + oldint(i,..) = O.
c
c       axoflo,bxoflo-hold values of aax & bbx (as defined in propsx) at i=1.
c       axofhi,bxofhi-hold values of aax & bbx(as defined in propsx)at i=itot.
c       flxlx, flxhs - hold fluxes at i=1 and i=itot boundaries.
c       ibchix, ibclox - boundary condition specifier.  Note that these are
c        defined for each regime, not each species.
c        1 = flux b.c. if inflow or no-flow at a boundary, but outflow b.c. if
c        there is outflow.
c        Anything else = "forced outflow" which gives zero flux if there
c        turns out to be inflow.
c
c       Arrays for y-transport.
c
       common /ycomon/
     +    windy(imaxd,jmaxd,kmaxd),      diffuy(imaxd,jmaxd,kmaxd),
     +    aay(imaxd,jmaxd,kmaxd,lpymax), bby(imaxd,jmaxd,kmaxd,lpymax),
     +    ccy(imaxd,jmaxd,kmaxd,lpymax),
```

Table 6.3b, Continued

```
     +    ayoflo(imaxd,kmaxd,lpymax),    byoflo(imaxd,kmaxd,lpymax),
     +    ayofhi(imaxd,kmaxd,lpymax),    byofhi(imaxd,kmaxd,lpymax),
     +    flxly(imaxd,kmaxd,lmaxd),      flxhy(imaxd,kmaxd,lmaxd),
     +    ibchiy(lmaxd), ibcloy(lmaxd)
c
c      just like xcomon
c
c      Arrays for vertical transport.
c
       common /scomon/
     +    winds(imaxd,jmaxd,kmaxd),diffus(imaxd,jmaxd,kmaxd,lpsmax),
     +    vertcl(imaxd,jmaxd,kmaxd,lpsmax),
     +    aas(kmaxd,lpsmax),    bbs(kmaxd,lpsmax),   ccs(kmaxd,lpsmax),
     +    asoflo(lpsmax),       bsoflo(lpsmax),      csoflo(lpsmax),
     +    asofhi(lpsmax),       bsofhi(lpsmax),      csofhi(lpsmax),
     +    flxls(imaxd,jmaxd,lmaxd), flxhs(imaxd,jmaxd,lmaxd),
     +    ibchis(lmaxd), ibclos(lmaxd),
     +    jfog(imaxd,jmaxd,kmaxd), dbydz2(ijkmax),
     +    drain(kmaxd), dsnow(kmaxd), vrain(kmaxd), vsnow(kmaxd)
c
c       Much of this parallels xcomon.  However, since s/z integrations are
c       performed one column at a time, most of the computational variables
c       have no i-j dimensions.
c
c       diffus = vertical eddy diffusivity.  Note fourth dimension which
c        may be used if one does energy equation and has different diffusivity
c        for energy and trace substances.
c       vertcl = settling velocity for each regime.  Negative (or zero) except
c        for helium.
c       jfog=saturation flag-- jfog = 1 if air is saturated or supersaturated.
c       drain, dsnow = pseudo-diffusivity for rain and snow.
c       vrain, vsnow = vertical fall velocities for rain and snow.
c
        common /column/ rclm(kmaxd,l2maxd), windsclm(kmaxd),
     +    cairclm(kmaxd), difusclm(kmaxd,lpsmax),ptotclm(kmaxd),
     +    vertcclm(kmaxd,lpsmax), flxlsclm(lmaxd), flxhsclm(lmaxd)
c
c       z-chem-z integrations are performed one "column" at a time. During the
c       integration, variables are moved from their 3 or 4 dimensional arrays
c       into these smaller "_clm" arrays, then back at the integration end.
c       this reduces page faulting.
c
       common / print_com / nprint, dtprint, tprint, dtdisk, tdisk, itdisk
```

Table 6.3b, Continued

```
c       nprint = subroutine print is called from main program after
c       nprint calls to core
c       dtdisk = disk output at increments of dtdisk
c       tdisk = next disk output at this time
c       itdisk = O - no disk output
c            other - disk output occurs
c       Note - tdisk is updated as output occurs.  It can be
c       initialized to zero (get initial output) or non-zero values.
c
       common / filter / nfiltx, nfilty, nfilts,
     +     nfilsrch,nfiltset,nfiltitr,
     +     filfacx(lmaxd), filfacy(lmaxd), filfacs(lmaxd),
     +     mbndfilx, mbndfily, mbndfils
c
c       nfiltx = filtering in x direction will occur after this many
c        x-integrations.
c       filfacx(l) = x-direction filter factor for species l.  if filtering
c        is to be done, use O.1O.  for no filtering, use O.
c        (this may be unnecessary.  normally one would filter all or no
c        species in the x-direction.)
c       nfilty, filfacy - same but for y.
c       nfilts, filfacs - same but for s/z.  in this case, one might wish to
c        filter only the precipitation bound species.
c       mbndfilx, mbndfily, mbndfils - controls the additional filtering at
c        inflow boundaries.  if the value is positive, the additional
c        filtering applies at the respective coordinate boundary.
c
c
c       odeint stuff: see odeint routine for definitions.
c
       common cmin(lmaxd),ratesf(lmaxd), cerror(lmaxd),
     + cnn(l2maxd),dtsave(imaxd,jmaxd,kmaxd),ggen(lmaxd),gdec(lmaxd),
     + eps,dtmin,ncorr,iexp
c
        common / inflowcmn /
     +      rinflowlox(jmaxd,kmaxd,lmaxd),
     +      rinflowhix(jmaxd,kmaxd,lmaxd),
     +      rinflowloy(imaxd,kmaxd,lmaxd),
     +      rinflowhiy(imaxd,kmaxd,lmaxd),
     +      rinflowlos(imaxd,jmaxd,lmaxd),
     +      rinflowhis(imaxd,jmaxd,lmaxd)
```

Table 6.3b, Continued

```
c       rinflowlox(j,k,l) = inflow mixing ratio of species l at grid 1,j,k
c       rinflowhix(j,k,l) = inflow mixing ratio of species l at grid itot,j,k
c       rinflowloy(i,k,l) = inflow mixing ratio of species l at grid i,1,k
c       rinflowhiy(i,k,l) = inflow mixing ratio of species l at grid i,jtot,k
c       rinflowlos(i,j,l) = inflow mixing ratio of species l at grid i,j,1
c       rinflowhis(i,j,l) = inflow mixing ratio of species l at grid i,j,ktot
c
c       rinflowlos would only be used if the lower s/z boundary were not the
c        earth's surface and inflow across this boundary were possible. For
c        theta-c, rinflow--- is the inflow theta-c value.
```

Table 6.4a: Listing of Subroutine **inptgn** for Case Example A

```
      subroutine inptgn
c
c     Class I subroutine called from program pluvius
c
c     Version for Case Example A
c
c     Inptgn initializes species names and indices, and grid indices. It also
c     sets a number of control variables, including subroutine bypass
c     controls, printing and time limits, and controls for execution of the
c     odeint ordinary differential-equation integrator.
      include 'pluvius.com'
c
c       Set initialization mode initmode.
c        If initmode = O compute initial conditions.
c        If initmode = 1 initialize by reading historical data from disk.
c       Historical data can be modified within this subroutine, if desired.
c
       initmode=O  ! Don't read from disk file for initialization.
c
c     Set dimensionality of integration domain.
c
      mtrans =1
```

Table 6.4a, Continued

```
      mtrany =1
      mtranx =O
c
c     set grid limits
c
      itot=1           !number of x grid points
      jtot=31          !number of y grid points
      ktot=35          !number of s (z) grid points
      ltot=4           !number of computed variables
      ltot2=7          !number of computed + constrained variables
      lpstot=4         !number of transport regimes in vertical dxn
      lpytot=2   !number of transport regimes in horizontal dxn
c
c     Set dependent variable indicies.
c
      lvapcl=1
      lrain=2
      lsnow=3
      ltheta=4
      lcloud=5
      lwvap=6
      ltemp=7
c
c     Simulation title and species names (character variables, used for
c     printout only).
c
      mtitle = ' users manual run: met only '
      name(lvapcl) = 'vap + cld'
      name(lrain) = 'rain'
      name(lsnow) = 'snow'
      name(ltheta) = 'theta'
      name(lcloud) = 'cloud'
      name(lwvap) = 'w vapor'
      name(ltemp) = 'temp'
c
c     Define indices for transport regimes.
c
      do 1O l=1,ltot2
        lprops(l)=2
        lpropy(l)=2
 1O   continue
      lprops(ltheta)=1
```

Table 6.4a, Continued

```
      lprops(lvapcl)=2
      lprops(lrain)=3
      lprops(lsnow)=4
      lprops(lcloud)=2
      lprops(lwvap)=2
      lprops(ltemp)=2
      lpropy(ltheta)=1
      lpropy(lvapcl)=2
      lpropy(lrain)=2
      lpropy(lsnow)=2
      lpropy(lcloud)=2
      lpropy(lwvap)=2
      lpropy(ltemp)=2
c
c     Set disk output controls:
c
c     print control for primary data output file "diskoutpt":
c
      nprint=480  !write every 7200 sec @ 2*dt=15 sec
c
c     print controls for unformatted backup file "dskbackup":
c
      itdisk=2   ! disk output: 1=continuous; 2=single updated frame
      tdisk= 3600.     !start backup disk at 3600 sec
      dtdisk= 3600.    !redo backup disk at 3600 sec increments
c
c     Set time spacing and limit.
      t=0.
      dt=7.5
      tstop= 36000.
c
c     Set spatial domain and grid.
c
c     Set spatial grids.
c
c     dsunif, dxunif, dyunif = grid lengths for uniform grid; IF A GRID IS
c     NONUNIFORM, SET ITS d_unif VALUE TO ZERO. Also if a grid is nonuniform,
c     set its array  _cord here. Otherwise the array(s) will be set in call
c     to subroutine grdxyz.
c
      dxunif = 0.
      dsunif = 15000.
      dyunif = 1375000.
```

Table 6.4a, Continued

```
      scord(1) = 0.
      ycord(1) = 0.
c
c     grdxyz sets the grid positions xcord, ycord, and scord for the case of
c     a uniform grid.
c
       call grdxyz(dxunif,dyunif,dsunif)
c
c     Set o.d.e. integrator (odeint) parameters (see odeint for definitions).
c
      eps=0.01
      dtmin=1.e-10
      ncorr=1
      iexp=1
      do 30 l=1,lmaxd
        ratesf(l)=1.e-2
        cmin(l)=1.e-24
 30   continue
      cmin(ltheta) = 1.e-8
      cmin(lvapcl) = 1.e-17
      cmin(lrain) = 1.e-17
      cmin(lsnow) = 1.e-17
      do 40 k = 1, ltot
        cerror(k) = cmin(k)*1.e4
 40   continue
c     Set boundary conditions: 1 = flux b.c. if inflow or no-flow at a
c     boundary, but outflow  b.c. if there is outflow.
c     Anything else = "forced outflow" which gives zero-gradient under inflow
c     conditions.
c
      do 50 l=1,ltot
        ibchis(l)=1
        ibclos(l)=1
        ibchiy(l)=1
        ibcloy(l)=1
 50   continue
c
c     Set subroutine calling frequencies.
c
      mfreqpy=2         !called on all occasions
      mfreqps=20        !called on all occasions
      mgen = 1          !call gen during all occasions
```

Table 6.4a, Continued

```
      mtemp=1          !energy balance integrated
      nonsym=-1        !symmetric integration performed
c
c     Set filter controls.
c
      nfilts = 1       !filter calls at each integration step
      nfilty = 1
      nfilsrch = 4
      nfiltset = 2
      nfiltitr = 3
c
c     Turn on filters for all species.
c
      do 60 l=1,ltot
        filfacs(l)=0.1
        filfacy(l)=0.1
 60   continue
c
c     boundary filters
c
      mbndfilx=-1
      mbndfily=1
      mbndfils=1
      return
      end
```

Table 6.4b: Listing of Subroutine **inptgn** for Case Example B

```
      subroutine inptgn
c
c     Class I subroutine called from program pluvius
c
c     Version for Case Example B
c
c     Inptgn initializes species names and indices, and grid indices. It
c     also sets a number of control variables, including subroutine bypass
c     controls, printing and time limits, and controls for execution of the
c     odeint ordinary differential-equation integrator.
c
```

Table 6.4b, Continued

```
       include 'pluvius.com'
c
c      Set initialization mode initmode.
c        If initmode = O compute initial conditions.
c        If initmode = 1 initialize by reading historical data from disk.
c       Historical data can be modified within this subroutine, if desired.
c
       initmode=O   ! Don't read from disk file for initialization.
c
c
c      Set dimensionality of integration domain.
c
       mtrans =1
       mtrany =1
       mtranx =O
c
c      set grid limits
c
       itot=1        !number of x grid points
       jtot=31       !number of y grid points
       ktot=35       !number of s (z) grid points
       ltot= 15      !number of computed variables
       ltot2= 22     !number of computed + constrained variables
       lpstot=4      !number of transport regimes in vertical dxn
       lpytot=2      !number of transport regimes in y-dxn
c
c      Set dependent variable indices.
c
       lso2gcl = 1
       lso2rain  = 2
       lso2snow  = 3
       lso4g = 4
       lso4gcl  = 5
       lso4rain =6
       lso4snow =7
       lno3gcl =8
       lno3g =9
       lno3rain =1O
       lno3snow =11
       lh2o2gcl =12
       lh2o2rain =13
       lh2o2snow =14
```

Table 6.4b, Continued

```
       lozone =15
       lcloud = 16
       lrain = 17
       lsnow = 18
       ltemp = 19
       lhcloud = 20
       lhrain = 21
       lvapcl = 22
c
c      Simulation title and species names (character variables, used for
c      printout only).
c
       mtitle = ' user manual run: chemistry simulation '
       name(lso2gcl) = 'g+c so2'
       name(lso2rain) = 'rain so2'
       name(lso2snow) = 'snow so2'
       name(lso4g) = 'g so4'
       name(lso4gcl) = 'g+c so4'
       name(lso4rain) = 'rain so4'
       name(lso4snow) = 'snow so4'
       name(lno3gcl) = 'g+c no3'
       name(lno3g) = 'g no3'
       name(lno3rain) = 'rain no3'
       name(lno3snow) = 'snow no3'
       name(lh2o2gcl) = 'g+c h2o2'
       name(lh2o2rain) = 'rain h2o2'
       name(lh2o2snow) = 'snow h2o2'
       name(lozone) = 'ozone'
       name(ltot+1) = 'cloud'
       name(ltot+2) = 'rain'
       name(ltot+3) = 'snow'
       name(ltot+4) = 'temp'
       name(ltot+5) = 'cloud h+'
       name(ltot+6) = 'rain h+'
       name(ltot+7) = 'c+w vap'
c
c      Define indices for transport regimes: regime indices arbitrarily chosen
c      here to be consistent with Case Example A, which solved energy equation
c      and used index 1 for theta.
c
       do 20 l=1,ltot2
         lpropy(l)=2
          lprops(l)=2
```

Table 6.4b, Continued

```
 20    continue
       lprops(lso2gcl)=2
       lprops(lso2rain)=3
       lprops(lso2snow)=4
       lprops(lso4g)=2
       lprops(lso4gcl)=2
       lprops(lso4rain)=3
       lprops(lso4snow)=4
       lprops(lno3g)=2
       lprops(lno3gcl)=2
       lprops(lno3rain)=3
       lprops(lno3snow)=4
       lprops(lh2o2gcl)=2
       lprops(lh2o2rain)=3
       lprops(lh2o2snow)=4
       lprops(lozone)=2
       lprops(lsnow)=4
       lprops(lrain)=3
c
c      Set disk output controls:
c
c      print control for primary data output file "dskoutpt":
c
       nprint=480  !write every 7200 sec @ 2*dt=15 sec
c
c      print controls for unformatted backup file "dskbackup"
c
       itdisk=2    ! disk output: 1=continuous; 2=single updated frame
       tdisk= 3600.      !start backup disk at 3600 sec
       dtdisk= 3600.     !redo backup disk at 3600 sec increments
c
c                 Set time spacing and limit.
c
       t=0.
       dt=7.5
       tstop= 36000.
c
c      Set spatial domain and grid.
c
c      Set spatial grids.
c
c      dsunif, dxunif, dyunif = grid lengths for uniform grid; IF A GRID IS
```

Table 6.4b, Continued

```
c      NONUNIFORM, SET ITS d_unif VALUE TO ZERO. Also if a grid is nonuniform,
c      set its array  _cord here. Otherwise the array(s) will be set in call
c      to subroutine grdxyz.
c
       dxunif = 0.
       dsunif = 15000.
       dyunif = 1375000.
       scord(1) = 0.
       ycord(1) = 0.
c
c      grdxyz sets the grid positions xcord, ycord, and scord for the case of
c      a uniform grid.
c
       call grdxyz(dxunif,dyunif,dsunif)
c
c       Set o.d.e. integrator (odeint) parameters (see odeint for definitions).
c
        eps=0.01
        dtmin=1.e-10
        ncorr=1
        iexp=1
        do 30 l=1,ltot
          ratesf(l)=1.e-2
          cmin(l)=1.e-24
30     continue
        do 40 l = l, ltot
         cerror(l) = cmin(l)*1.e4
40     continue
c
c       Set boundary conditions:    1 = flux b.c. if inflow or no-flow at a
c        boundary, but outflow  b.c. if there is outflow.
c       Anything else = "forced outflow" which gives zero-gradient under inflow
c      conditions.
c
        do 50 l=1,ltot
          ibchis(l)=1
          ibclos(l)=1
          ibchiy(l)=1
          ibcloy(l)=1
50     continue
c
c       Set subroutine calling frequencies.
c
```

Table 6.4b, Continued

```
      mfreqpy=2        !called on all occasions
      mfreqps=20       !called on all occasions
      mgen = 1         !call gen during all occasions
      mijpack=1        !pack/unpack
      mtemp=0          !energy balance not integrated
      nonsym=-1        !symmetric integration performed
c
c     Set filter controls.
c
      nfilts = 1       !filter calls at each integration step
      nfilty = 1
      nfilsrch = 4
      nfiltset = 2
      nfiltitr = 3
c
c     Turn on filters for all species.
c
      do 60 l=1,ltot
        filfacs(l)=0.1
        filfacy(l)=0.1
 60   continue
C
c     boundary filters
c
      mbndfilx=-1
      mbndfily=1
      mbndfils=1
      return
      end
```

Table 6.5: Listing of Subroutine **femset**

```
       subroutine femset
       include 'pluvius.com'
c
c      Class II subroutine called from program pluvius
c
c      femset computes grid expansion factors and common grid terms for
c       subsequent use in numerical approximations.  Interrogated only once
c      during program execution.
c
       do 10 i = 1,ijkmax
         do 10 idum = 1,3
           ct1a(idum,i) = 0.
           ct1c(idum,i) = 0.
           ct3a(idum,i) = 0.
           ct3b(idum,i) = 0.
           ct3c(idum,i) = 0.
           ct3d(idum,i) = 0.
           ct3d(idum,i) = 0.
           ct3e(idum,i) = 0.
           ct3f(idum,i) = 0.
           ct4b(idum,i) = 0.
           ct4d(idum,i) = 0.
 10    continue
c
c      x coordinate
c
       if (itot.gt.1) then
         do 20 i = 1,itot-1
           deltax(i) = xcord(i+1) - xcord(i)
 20      continue
         do 30 i = 2,itot
           ct1a(1,i) = 1./deltax(i-1)
           ct3a(1,i) = deltax(i-1)/12.
           ct3b(1,i) = deltax(i-1)/6.
           ct3c(1,i) = deltax(i-1)/4.
           ct3d(1,i) = deltax(i-1)/3.
           ct4b(1,i) = deltax(i-1)/2.
30       continue
         do 40 i = 1,itot-1
           ct1c(1,i) = 1./deltax(i)
           ct3e(1,i) = deltax(i)/12.
           ct3f(1,i) = deltax(i)/3.
           ct3h(1,i) = deltax(i)/6.
```

Table 6.5, Continued

```
          ct4d(1,i) = deltax(i)/2.
          ct3df(1,i) = ct3d(1,i) + ct3f(1,i)
40      continue
        do 50 i = 2, itot-1
          fdcofx(1,i) = -deltax(i)/(deltax(i-1)*
     +    (deltax(i)+deltax(i-1)))
          fdcofx(2,i) = (deltax(i)-deltax(i-1))/
     +    (deltax(i)*deltax(i-1))
          fdcofx(3,i) = deltax(i-1)/(deltax(i)*
     +    (deltax(i)+deltax(i-1)))
50      continue
        ct3df(1,1) = ct3f(1,1)
        ct3df(1,itot) = ct3d(1,itot)
      end if
c
c     y coordinate
c
      if (jtot.gt.1) then
        do 60 j = 1,jtot-1
          deltay(j) = ycord(j+1)-ycord(j)
 60     continue
        do 70 j = 2,jtot
          ct1a(2,j) = 1./deltay(j-1)
          ct3a(2,j) = deltay(j-1)/12.
          ct3b(2,j) = deltay(j-1)/6.
          ct3c(2,j) = deltay(j-1)/4.
          ct3d(2,j) = deltay(j-1)/3.
          ct4b(2,j) = deltay(j-1)/2.
 70     continue
        do 80 j = 1,jtot-1
          ct1c(2,j) = 1./deltay(j)
          ct3e(2,j) = deltay(j)/12.
          ct3f(2,j) = deltay(j)/3.
          ct3h(2,j) = deltay(j)/6.
          ct4d(2,j) = deltay(j)/2.
          ct3df(2,j) = ct3d(2,j) + ct3f(2,j)
 80     continue
        do 90 j = 2, jtot-1
          fdcofy(1,j) = -deltay(j)/(deltay(j-1)*
     +    (deltay(j)+deltay(j-1)))
          fdcofy(2,j) = (deltay(j)-deltay(j-1))/
     +    (deltay(j)*deltay(j-1))
```

Table 6.5, Continued

```
          fdcofy(3,j) = deltay(j-1)/(deltay(j)*
     +    (deltay(j)+deltay(j-1)))
 90     continue
        ct3df(2,1) = ct3f(2,1)
        ct3df(2,jtot) = ct3d(2,jtot)
      end if
c
c     sigma (or z) coordinate
      if (ktot .gt. 1) then
        do 100 k = 1,ktot-1
          deltas(k) = scord(k+1)-scord(k)
 100    continue
        do 110 k = 2,ktot
          ct1a(3,k) = 1./deltas(k-1)
          ct3a(3,k) = deltas(k-1)/12.
          ct3b(3,k) = deltas(k-1)/6.
          ct3c(3,k) = deltas(k-1)/4.
          ct3d(3,k) = deltas(k-1)/3.
          ct4b(3,k) = deltas(k-1)/2.
 110    continue
        do 120 k = 1,ktot-1
          ct1c(3,k) = 1./deltas(k)
          ct3e(3,k) = deltas(k)/12.
          ct3f(3,k) = deltas(k)/3.
          ct3h(3,k) = deltas(k)/6.
          ct4d(3,k) = deltas(k)/2.
          ct3df(3,k) = ct3d(3,k) + ct3f(3,k)
 120    continue
        do 130 k = 2, ktot-1
          fdcofs(1,k) = -deltas(k)/(deltas(k-1)*
     +    (deltas(k)+deltas(k-1)))
          fdcofs(2,k) = (deltas(k)-deltas(k-1))/
     +    (deltas(k)*deltas(k-1))
          fdcofs(3,k) = deltas(k-1)/(deltas(k)*
     +    (deltas(k)+deltas(k-1)))
 130    continue
        ct3df(3,1) = ct3f(3,1)
        ct3df(3,ktot) = ct3d(3,ktot)
      end if
      ct2a = 1./3.
      ct2c = 1./6.
      return
      end
```

Table 6.6a: Listing of Subroutine **init** for Case Example A

```
      subroutine init
c
c     Class I subroutine called from program pluvius
c
c
c     init computes initial values r(i,j,k,l) of the dependent variables for the
      entire solution matrix.
c
c     This version of init depicts an idealized warm front.
c
      include 'pluvius.com'
      dimension surft(imaxd,jmaxd),surfp(imaxd,jmaxd)
      data alpha /3.526e-4/ !deg k/cm
c
c     The following block synthesizes the values based on stipulated surface
c     conditions and assumed concentration profiles.  This will be
c     unnecessary if real data are available.
c
c     First set surface pressures, temperatures, concentrations, and mixing
c     ratios.
c
c     Case for preset of met-only computations:  set i.c.'s to left of front.
c
      surft(1,1)=290.        !  deg k
      surfp(1,1)=1.013e6     !  dyne / cm2 units
      jfog(1,1,1)=0
c
c     cloud flag: jfog = 0 for clear air, =1 for condensation
c
      r(1,1,1,ltemp)=surft(1,1)
      cair(1,1,1)=aircon(surfp(1,1),surft(1,1)) !moles air/cc
      r(1,1,1,lvapcl)=.65*sat(surft(1,1))/cair(1,1,1)
      ptot(1,1,1)=surfp(1,1)
      rr=sat(r(1,1,1,ltemp))/cair(1,1,1)
c
c     mixing ratio of water in saturated air
c
      r(1,1,1,lcloud)=dim(r(1,1,1,lvapcl),rr)
      r(1,1,1,lwvap)=r(1,1,1,lvapcl)-r(1,1,1,lcloud)
      jfog(1,1,1)=.9999999+r(1,1,1,lcloud)
c
c     Now estimate air concentrations from approximate integrated
```

Table 6.6a, Continued

```
c      hydrostatic equation  (j=1 column only).
c
       do 20 k=2,ktot
         cair(1,1,k)=cair(1,1,1)*exp(-1.66e-6*scord(k))
c
c      Now compute temperature profiles based on wet or dry standard lapses.
c
         r(1,1,k,ltemp)=r(1,1,k-1,ltemp)+rlapse(r(1,1,k-1,ltemp),
    +     cair(1,1,k-1),jfog(1,1,k-1))*deltas(k-1)
c
c      Set humidities and local condensed water flag.
         do 10 m=1,3
           r(1,1,k,lvapcl)=.65*sat(r(1,1,k,ltemp))/cair(1,1,k)
           floatk = float(-k+20)
           if(k.gt.20) r(1,1,k,lvapcl) = r(1,1,k,lvapcl)*exp(floatk)
c
c          (Set 65% rh up to k=20 and decreasing aloft.)
c
           rr=sat(r(1,1,k,ltemp))/cair(1,1,k)
           r(1,1,k,lcloud)=dim(r(1,1,k,lvapcl),rr)
           r(1,1,k,lwvap)=r(1,1,k,lvapcl)-r(1,1,k,lcloud)
           jfog(1,1,k)=.9999999+r(1,1,k,lcloud)
           cair(1,1,k)=cair(1,1,k-1)*
     +      exp((-alpha-rlapse(r(1,1,k-1,ltemp),
     +      cair(1,1,k-1),jfog(1,1,k)))*deltas(k-1)/
     +      (0.5*(r(1,1,k-1,ltemp)+r(1,1,k,ltemp))))
 10      continue
         ptot(1,1,k)=cair(1,1,k)*glc*r(1,1,k,ltemp)
 20    continue
c
c      Now paint j=1 column into interior grid points to right: initialization
c      array only.
c
       do 30 j=2,jtot
         do 30 k=1,ktot
           do 25 l=1,ltot2
             r(1,j,k,l)=0.
 25        continue
           jfog(1,j,k)=0
           ptot(1,j,k) = ptot(1,1,k)
           cair(1,j,k)=cair(1,1,k)
           r(1,j,k,lcloud)=r(1,1,k,lcloud)
           r(1,j,k,lwvap)=r(1,1,k,lwvap)
```

Table 6.6a, Continued

```
          r(1,j,k,lvapcl)=r(1,1,k,lvapcl)
          r(1,j,k,ltemp)=r(1,1,k,ltemp)
 30   continue
c
c     Now set right-hand b.c.
c
c     Case for preset of met-only computations:  set i.c.'s to right of front
c     j=jtot column only.
c
      surft(1,jtot)=282.
      surfp(1,jtot)=1.013e6   !  dyne / cm2 units
      jfog(1,1,jtot)=0
      r(1,jtot,1,ltemp)=surft(1,jtot)
      cair(1,jtot,1)=surfp(1,jtot)/(8.31e7*surft(1,jtot))
      r(1,jtot,1,lvapcl)=.45*sat(surft(1,jtot))/cair(1,jtot,1)
      ptot(1,jtot,1)=surfp(1,jtot)
      rr=sat(r(1,jtot,1,ltemp))/cair(1,jtot,1)
      r(1,jtot,1,lcloud)=dim(r(1,jtot,1,lvapcl),rr)
      r(1,jtot,1,lwvap)=r(1,jtot,1,lvapcl)-r(1,jtot,1,lcloud)
      jfog(1,jtot,1)=.9999999+r(1,jtot,1,lcloud)
c
c     Now estimate air concentrations from approximate integrated hydrostatic
c     equation (just for lower portion of cold side).
c
      do 45 k=2,15
        cair(1,jtot,k)=cair(1,jtot,1)*exp(-1.66e-6*scord(k))
c
c     Compute temperature profiles based on wet or dry
c     standard lapses.
c
        r(1,jtot,k,ltemp)=r(1,jtot,k-1,ltemp)+
     +    rlapse(r(1,jtot,k-1,ltemp),cair(1,jtot,k-1),
     +    jfog(1,jtot,k-1))*deltas(k-1)
c
c     Set humidities and local condensed water flag.
c
        do 40 m=1,3
          r(1,jtot,k,lvapcl)=.45*sat(r(1,jtot,k,ltemp))/cair(1,jtot,k)
c     (Set 45% rh up to top of cold wedge.)
          rr=sat(r(1,jtot,k,ltemp))/cair(1,jtot,k)
          r(1,jtot,k,lcloud)=dim(r(1,jtot,k,lvapcl),rr)
          r(1,jtot,k,lwvap)=r(1,jtot,k,lvapcl)-r(1,jtot,k,lcloud)
```

Table 6.6a, Continued

```
          jfog(1,jtot,k)=.9999999+r(1,jtot,k,lcloud)
          cair(1,jtot,k)=cair(1,jtot,k-1)*
     +      exp((-alpha-rlapse(r(1,jtot,k-1,ltemp),
     +      cair(1,jtot,k-1),jfog(1,jtot,k)))*deltas(k-1)/
     +      (0.5*(r(1,jtot,k-1,ltemp)+r(1,jtot,k,ltemp))))
 40     continue
        ptot(1,jtot,k)=cair(1,jtot,k)*glc*r(1,jtot,k,ltemp)
 45   continue
c
c     Now set initial conditions to right of and below front.
c
      do 50 j= 7,jtot-1
        do 50 k=1,15
          jj=j-7
          if(k.gt.jj) go to  50
          r(1,j,k,ltemp)=r(1,jtot,k,ltemp)
          r(1,j,k,lvapcl)=r(1,jtot,k,lvapcl)
          r(1,j,k,lcloud)=r(1,jtot,k,lcloud)
          r(1,j,k,lwvap)=r(1,j,k,lwvap)
 50   continue
c
c     Now compute final pressures, concentrations, and thetas across complete
c     domain.  Note that ptot does not change from this point on.
c
      do 60 j=1,jtot
        r(1,j,1,ltheta)=(1.e6/ptot(1,1,1))**ak*r(1,j,1,ltemp)*
     +    (1.-alocp*r(1,j,1,lcloud)/r(1,j,1,ltemp))
        ptot(1,j,1)=ptot(1,1,1)
        cair(1,j,1)=ptot(1,j,1)/(glc*r(1,j,1,ltemp))
        do 60 k=2,ktot
          ptot(1,j,k)=ptot(1,j,k-1)*(1.-deltas(k-1)*alpha/r(1,j,k-1,ltemp))
          r(1,j,k,ltheta)=(1.e6/ptot(1,j,k))**ak*r(1,j,k,ltemp)*
     +      (1.-alocp*r(1,j,k,lcloud)/r(1,j,k,ltemp))
          cair(1,j,k)=ptot(1,j,k)/(glc*r(1,j,k,ltemp))
 60   continue
c
c     Activate disk read if required.
c
      if(initmode.eq.1) call restart(trestart)
c
c
c     Finally, set the (time-invariant) inflow concentrations; these will be
c     used in the boundary-condition subroutines to determine inflow rates.
```

Table 6.6a, Continued

```
c      This should be modified if boundary conditions vary with time.
c
       call inflowinit
       return
       end
```

Table 6.6b: Listing of Subroutine **`init`** for Case Example B

```
       subroutine init
c
c      Class I subroutine called from program pluvius
c
       include 'pluvius.com'
c
c      Computes initial values r(i,j,k,l) of the dependent variables for the
c      entire solution matrix.  For this application, the meteorological
c      variables will be supplied through the call of subroutine cloudinit,
c      which appears immediately below; thus the remainder of subroutine init
c      will simply compute some of the ancillary variables, plus the chemical
c      variables, and return.
c
c      Read meteorological input from disk storage.
c
       call cloudinit
c
c      Set cloud flags.
c
       do 10 j=1,jtot
         do 10 k=1,ktot
           jfog(1,j,k)=0
           if(r(1,j,k,lcloud).gt.0.) jfog(1,j,k)=1
c
c      Compute pressures using equation of state.
c
           ptot(1,j,k)=cair(1,j,k)*glc*r(1,j,k,ltemp)
 10    continue
c
```

Table 6.6b, Continued

```
c      Input initial pollutant mixing ratios.
c
c      Zero internal concentrations.
c
       do 20 j=1,jtot
         do 20 k=1,ktot
           do 20 l=1,15
             r(1,j,k,l)=0.
 20    continue
c
c      Now set mixing ratios throughout.
c
       do 30 k=1,ktot
         do 30 j=1,jtot
           if(scord(k).lt.150000) then !below 1500 m layer
             r(1,j,k,lso2gcl) = 1.e-8
             r(1,j,k,lso4g) = 1.e-9
             r(1,j,k,lso4gcl) = 1.e-9
             r(1,j,k,lno3gcl) = 1.e-9
             r(1,j,k,lno3g) = .2e-9
             r(1,j,k,lh2o2gcl) = 1.e-9
             r(1,j,k,lozone) = 5.e-8
           else !above 1500 m layer
             rr=exp(.00001*(150000.-scord(k)))
             r(1,j,k,lso2gcl) = 1.e-8*rr
             r(1,j,k,lso4g) = 1.e-9*rr
             r(1,j,k,lso4gcl) = 1.e-9*rr
             r(1,j,k,lno3gcl) = 1.e-9*rr
             r(1,j,k,lno3g) = .2e-9*rr
             r(1,j,k,lh2o2gcl) = 1.e-9
             r(1,j,k,lozone) = 5.e-8
           end if
 30    continue
c
c      Activate disk read if required.
c
       if(initmode.eq.1) call restart(trestart)
c
C
c      Finally, set the (time-invariant) inflow concentrations; these will be
c      used in the boundary-condition subroutines to determine inflow rates.
c      This should be modified if boundary conditions vary with time.
```

Table 6.6b, Continued

```
      call inflowinit
      return
      end
```

Table 6.7: Listing of Subroutine **wind**

```
      subroutine wind
c
c         Class I subroutine called from program pluvius
c
c      wind generates the matrix of u,v, and w (or sigma) wind components that
c      are stored in the arrays windx, windy, and winds.  This particular 2-d
c      version is an improved wind field that was produced after the
c      Hales (1989) Atmospheric Environment article.
c
      include 'pluvius.com'
      data ifirst / 1 /
c
c      statement functions for "first-guess" molar air density and for
c      vertical velocity
c
      cc(z)=4.308e-5*((283.-9.8e-5*z)/283.)**2.5
      ww(z,ztop,wtop)=wtop*(z/ztop)**1.5
c
      if(ifirst.eq.0) return
      ifirst = 0
      jbase = 3
c
c      Do inflow block to left of front:
c
      do 10 k=1,ktot
        zz=(k-1)*15000.
        cdum=cc(zz)    !internal function to compute molar air density
        windy(1,1,k)=0.0345/cdum
```

Table 6.7, Continued

```
          winds(1,1,k)=0.
          do 10 j=2,jbase-1
            windy(1,j,k)=windy(1,1,k)
            winds(1,j,k)=winds(1,1,k)
 10     continue
c
c       do block above frontal slope:
c
        do 20 j=jbase,jbase+12
          jj=j-jbase+2  !jj = k index of element just above frontal surface
          height=(jj-2)*15000.
          slope =.0109
          if(j.eq.jbase) slope=slope/2
          do 20 k=jj,ktot
            zz=(k-1)*15000.
            airc=cc(zz)
            zzero=510000.*zz/(510000.-height)
            airczero=cc(zzero)
            windyzero=.0345/airczero
            windy(1,j,k)=windyzero*airczero*510000./
     +      (airc*(510000.-height))
          winds(1,j,k)=windy(1,j,k)*slope*(510000.-zz)/
     +      (510000.-height)
 20     continue
c
c       Fill in winds just on frontal surface (ident. to those just above).
c
        do 30 j=jbase,jbase+13
          jj=j-jbase+1
          windy(1,j,jj)=windy(1,j,jj+1)
          winds(1,j,jj)=winds(1,j,jj+1)
 30     continue
        winds(1,jbase,1)=0.
c
c       Now go for inner wedge.
c
        do 40 k=1,12
          do 40 j=jbase+k,jbase+13
            ktop=j-jbase+1          ! ktop = k index on frontal line
            ztop=(ktop-1)*15000.
            if(k.ne.1) then
              zzminus=(k-2)*15000.
              zzplus=k*15000.
```

Table 6.7, Continued

```
            cminus=cc(zzminus)
            cplus=cc(zzplus)
            wminus=ww(zzminus,ztop,winds(1,j,ktop))
            wplus=ww(zzplus,ztop,winds(1,j,ktop))
            deriv=(cplus*wplus-cminus*wminus)/30000.
          else
            cplus=cc(15000.)
            wplus=ww(15000.,ztop,winds(1,j,ktop))
            deriv=cplus*wplus/15000.
          end if
          zz=(k-1)*15000.
          cstar=cc(zz)
          winds(1,j,k)=ww(zz,ztop,winds(1,j,ktop))
          windy(1,j,k)=windy(1,j-1,k)-1375000.*deriv/cstar
 40     continue
c
c       Do right hand side of storm.
c
        do 50 j=jbase+13,jbase+18
          floatj=j
          float=jbase+18
          do 50 k=1,ktot
          winds(1,j,k)=((float-floatj)/6.)*winds(1,jbase+12,k)
 50     continue
        do 60 j=jbase+19,jtot
          do 60 k=1,ktot
            winds(1,j,k)=winds(1,jbase+18,k)
 60     continue
        do 70 k=1,ktot
          sum=0.
          zz=(k-1)*150.
          cstar=cc(zz)
          zzplus=zz+15000.
          zzminus=zz-15000.
          if(k.ne.ktot) cstarplus=cc(zzplus)
          if(k.ne.1) cstarminus=cc(zzminus)
          do 70 j=jbase+13,jtot
            if(k.ne.1.and.k.ne.35) then
              deriv=(winds(1,j,k+1)*cstarplus-winds(1,j,k-1)
     +        *cstarminus)/30000.
            else
              if(k.eq.1) deriv=(winds(1,j,2)*cstarplus-
```

Table 6.7, Continued

```
     +       winds(1,j,1)*cstar)/15000.
             if(k.eq.35) deriv=(winds(1,j,35)*cstar-
     +       winds(1,j,34)*cstarminus)/15000.
           end if
           sum=sum+deriv*1370000.
           windy(1,j,k)=(windy(1,jbase+12,k)*cstar-sum)/cstar
 70    continue
       return
       end
```

Table 6.8: Listing of Subroutine **printi**

```
      subroutine printi
c
c      Class I subbbroutine called from program pluvius
c
c
c      Opens output files ande writes selected input information and initial-
c      condition arrays to file diskoutpt.
c
       include 'pluvius.com'
       open(unit=27, file='diagnostics',status='unknown')
       open(unit=20, file='diskoutpt', status='new')
       ncolumns=11
       do 10 l=1,ltot2
        write(20,2001) mtitle, name(l)
         do 20 j1=1,jtot,ncolumns
           j2=min(j1+ncolumns-1, jtot)
           do 30 k=ktot,1,-1
             write(20,2002) (r(1,j,k,l), j=j1,j2)
 30        continue
           write(20,2003)
 20      continue
 10    continue
 2001  format(1h1,//,a,//,' initial fields for ',a)
 2002  format (1h ,15(1x,e9.3))
 2003  format(1h1,//)
       return
       end
```

Table 6.9: Listing of Subroutine **`print`**

```
      subroutine print
c
c     Class I subroutine called from program pluvius
c
c     Handles routine printout of computed results to disk.
c
      include 'pluvius.com'
       ncolumns=11
      do 10l=1,ltot2
        write(20,2001) mtitle, name(l), t
        write(20,2004)
        do 20 j1=1,jtot,ncolumns
          j2=min(j1+ncolumns-1, jtot)
          do 30 k=ktot,1,-1
            write(20,2002) (r(1,j,k,l), j=j1,j2)
 30       continue
          write(20,2003)
 20      continue
 10    continue
 2001  format(1h1,//,a,//,' fields for ',a,' at time = ',e10.4,//)
 2002  format (1h ,15(1x,e9.3))
 2003  format(1h1,//,'-4444') !finder tag for subsequent data processing
 2004  format(1h ,'-9999')    !finder tag for subsequent data processing
       return
       end
```

Table 6.10: Listing of Subroutine **`matbal`**

```
       subroutine matbal
c
c      Class III subroutine called from program pluvius.
c
c     Subroutine to compute advective inflow/outflow rates at boundaries.
c      Present version assumes two-dimensional system, and computes rates
```

Table 6.10, Continued

```
c      flowing in horizontal(y) direction at left and right, plus rates at
c      surface.  This must be modified for 3-d systems.
c      Note that, because dry deposition is not computed in this example,
c      and also because the zero-gradient approximation is made at the y
c      boundaries, diffusive comnponents of the fluxes are not calculated
c      here.  This should be modified if dry deposition or surface emissions
c      are included in downstream applications.
c
       include 'pluvius.com'
       dimension fsl(ijkmax,l2maxd),flux(ijkmax),total(l2maxd)
c
c      Set print boundaries.
c
      ncolumns=11
c
c      do lhs
c
       do 5 l=1,ltot2
         do 4 k=1,ktot
           fsl(k,l) = windy(1,1,k)*r(1,1,k,l)*cair(1,1,k)
           flux(k)=fsl(k,l)
 4        continue
c
c      Integrate fluxes across k domain of lhs.
c
         call integrate(flux,scord,ktot,rturn)
         total(l)=rturn
 5     continue
c
c      output lhs values:
c
      do 10 l=1,ltot2
        write(20,*)
        write(20,*) "species = ", name(l)
        write(20,*) "LHS horizontal fluxes (moles/cm2-sec or deg./cm2-sec),
     +  for k=1 ... ktot"
        do 20 k1=1,ktot,ncolumns
          k2=min(k1+ncolumns-1, ktot)
          write(20,2002) (fsl(k,l), k=k1,k2)
 20     continue
 10    continue
       write (20,2003)
       do 50 l=1,ltot2
```

Table 6.10, Continued

```
          write(20,2004) l, total(l)
 50     continue
c
c       do rhs
c
        do 65 l=1,ltot2
          do 60 k=1,ktot
            fsl(k,l) = windy(1,jtot,k)*r(1,jtot,k,l)*cair(1,jtot,k)
            flux(k)=fsl(k,l)
 60       continue
c
c       Integrate fluxes across k domain on rhs>
c
          call integrate(flux,scord,ktot,rturn)
          total(l)=rturn
 65     continue
c
c       Output rhs statistics:
c
        do 70 l=1,ltot2
          write(20,*)
          write(20,*) "species = ", name(l)
          write(20,*) "RHS horizontal fluxes (moles/cm2-sec or deg/cm2-sec),
      +   for k=1 ... ktot"
          do 80 k1=1,ktot,ncolumns
            k2=min(k1+ncolumns-1, ktot)
            write(20,2002) (fsl(k,l), k=k1,k2)
 80       continue
 70     continue
        write (20,2007)
        do 100 l=1,ltot2
          write(20,2004) l, total(l)
 100    continue
c
c       Do bottom.
        do 110 j=1,jtot
c
c       Compute rain and snow fall velocities at surface.
c
         call ijpack(1,j)
         call transport(1,j)
         do 110 l=1,ltot2
```

Table 6.10, Continued

```
          fsl(j,l)=0.
          if(l.le.ltot) then
          if(lprops(l).eq.lprops(lrain)) then
          fsl(j,l)=vertcclm(1,lprops(lrain))*r(1,j,1,l)*cair(1,j,1)
          end if
          end if
 110   continue
       do 112 l=1,ltot2
         do 111 j=1,jtot
           flux(j)=fsl(j,l)
 111     continue
c
c      Integrate across j domain on grid bottom.
c
         call integrate(flux,ycord,jtot,rturn)
         total(l)=rturn
 112   continue
c
c      Output bottom statistics:
c
       do 120 l=1,ltot2
         write(20,*)
                    write(20,*) "species = ", name(l)
         write(20,*) "bottom fluxes (moles/cm2-sec),
     +   for j=1 ... jtot"
         do 130 j1=1,jtot,ncolumns
           j2=min(j1+ncolumns-1, jtot)
           write(20,2002) (fsl(j,l), j=j1,j2)
 130     continue
 120   continue
       write (20,2011)
       do 150 l=1,ltot2
         write(20,2004) l, total(l)
 150   continue
 2002  format(1h ,/,11(1x,e10.3))
 2003  format (1h ,/,'LHS amounts (moles/cm sec or deg/cm-sec)',//,
     + '   l    amount',/)
 2004  format(1h ,i3,1x,e10.4)
 2006  format(1h ,/,i5,10(1x,e10.3))
 2007  format (1h ,/,'RHS amounts (moles/cm sec or deg/cm-sec)',//,
     + '   l    amount',/)
```

Table 6.10, Continued

```
 2011  format (1h ,/,'bottom amounts (moles/cm sec)',//,
     + '   l    amount',/)
       return
       end
```

Table 6.11: Listing of Subroutine **diff**

```
       subroutine diff
c
c      Class I subroutine called from program pluvius
c
c      Provides eddy diffusivities in cm**2/s.
c
       include 'pluvius.com'
       data ifirst /1/
       if (ifirst .ne.1) return
       ifirst = -1
       do 100 k = 1, ktot
         do 100 j = 1, jtot
           do 100 i = 1, itot
             diffuy(i,j,k) = 1.0e5
             diffus(i,j,k,1) = 1.0e5
             diffus(i,j,k,2) = 1.0e5
             diffus(i,j,k,3) = 1.0e5
             diffus(i,j,k,4) = 1.0e5
100    continue
       return
       end
```

Table 6.12: Listing of Subroutine **core**

```
       subroutine core
c
c      Class II subroutine called from program pluvius
c
c      Coordinates numerical integration of governing equations for a single
c     or double time increment, dt ot 2*dt.  The choice between a single or
c     double step is controlled by the variable nonsym (>O for single step;
c     = O for double step), which is set in subroutine inputgn,  a double
c     step (symmetrical integration) is recommended.  This integration process
c     proceeds as follows:  integration in x for dt; integration in y for dt;
c     integration in s for dt;  integration of transformation terms for 2*dt;
c     integration in s for dt;  integration in y for dt;  integration in x
c     for dt.
c
      include 'pluvius.com'
      data ifirst/1/
      dtcur = dt
      if (ifirst .eq. 1) call coreOO
      ifirst = O
c
c     Stage 1: initial integration in x (over time dt).
c
      if ((itot.gt.1) .and. (mtranx.gt.O)) then
        trun = t+dt
        call xinteg( 1 )
      endif
c
c     Stage 2: initial integration in y (over time dt).
c
      if ((jtot.gt.1) .and. (mtrany.gt.O)) then
       trun = t+dt
       call yinteg( 1 )
      end if
c
c     Stage 3: (possible) subincrementation of dt into an integral number
c     of steps having length dtsub.  Initial integration in sigma direction
c     over period dtsub, composite integration of kinetics step over period
c     2*dtsub, and second integration in sigma direction over next dtsub
c     increment.  Repeat until major time step dt is attained.
c
      tOsave = t
      do 5O j = 1, jtot
```

Table 6.12, Continued

```
        do 5O i = 1, itot
          call ijpack( i, j )
c
c     Evaluate subincrement size dtsub (dt/dtsub must be an integer).
c
          call sbstep( dtsub, nsubt, i, j)
          dtcur = dtsub
          dtchem = 2.O*dtsub
          if (nonsym .gt. O) dtchem = dtsub
c
c     Set filter counters for start of sub integration loops.
c
          if ((ktot.gt.1) .and. (mtrans.gt.O)) call sfilter(-1,O,O,O)
c
c      Loop for each sub integration
c
          do 5O isubt = 1,nsubt
c
c      initial subintegration in sigma
c
            trun = tOsave + (2*isubt-1)*dtsub
            if (nonsym .gt. O) trun = tOsave + isubt*dtsub
            t = trun - dtsub
            if ((ktot.gt.1) .and. (mtrans.gt.O)) then
              call sinteg( 1, isubt, i, j )
            end if
c
c      composite integration of kinetics step over time 2.*dtsub
c
            if (mgen .gt. O) then
              call chmint(i, j, t, dtchem )
            end if
c
c      secondary integration in sigma direction
c
            if (nonsym .gt. O) goto 5O
            trun = tOsave + (2*isubt)*dtsub
            t = trun-dtsub
            if ((ktot.gt.1) .and. (mtrans.gt.O)) then
              call sinteg( 2, isubt, i, j )
            end if
5O    continue
```

Table 6.12, Continued

```
      if ((ktot.gt.1) .and. (mtrans.gt.0)) call sfilter( -2, 0, 0, 0 )
      call ijunpack
c
c     end stage 3
c
       t = t0save + dt
       dtcur = dt
       if (nonsym .gt. 0) goto 90
c
c      Stage 4:  secondary integration in y over time step dt.
c
       if ((jtot.gt.1) .and. (mtrany.gt.0)) then
         trun = t+dt
         call yinteg( 2 )
       end if
c
c     Stage 5:  secondary integration in x over time step dt.
c
       if ((itot.gt.1) .and. (mtranx.gt.0)) then
         trun = t+dt
         call xinteg( 2 )
       end if
       t = t+dt
90     continue
       return
       end
```

Table 6.13a: Listing of Subroutine **cleanup** for Case Example A

```
      subroutine cleanup
      include 'pluvius.com'
c
c     Class I subroutine called from program pluvius
c
c     User-supplied subroutine to remove extraneous phases and associated
c     pollutants, as well as to reestablish values of constrained variables
```

Table 6.13a, Continued

```
c      after each time step.
c
c      This version is for the met-only demo run for the user's manual
c      (Case Example A).
c
       if(mtemp.eq.1) then
         do 20 j=1,jtot
           do 20 k=1,ktot
c      Calculate temperatures from theta-c's.
             call tfrmth(ptot(1,j,k),r(1,j,k,ltheta),
    +        r(1,j,k,lvapcl),jfog(1,j,k),r(1,j,k,ltemp))
             r(1,j,k,lcloud)=r(1,j,k,lvapcl)-
     +       sat(r(1,j,k,ltemp))/cair(1,j,k)
             if(r(1,j,k,lcloud).le.0.) jfog(1,j,k)=0
             r(1,j,k,lcloud)=amax1(r(1,j,k,lcloud),0.)
             r(1,j,k,lwvap)=r(1,j,k,lvapcl)-r(1,j,k,lcloud)
20       continue
       end if
       do 10 j=1,jtot
         do 10 k=1,ktot  !Eliminate any negative mixing ratios.
           r(1,j,k,lrain)=amax1(r(1,j,k,lrain),0.)
           r(1,j,k,lsnow)=amax1(r(1,j,k,lsnow),0.)
           if(r(1,j,k,ltemp).gt.273.2.and.r(1,j,k,lsnow).lt.1.e-5)
     +     then   !Melt residual snow if above freezing.
             r(1,j,k,lrain)=r(1,j,k,lrain)+r(1,j,k,lsnow)
             r(1,j,k,lsnow)=0.
           end if
         if(r(1,j,k,lcloud).le.0..and.r(1,j,k,lsnow).lt.1.e-6)
     +   then     !Sublime residual snow if air subsaturated.
           r(1,j,k,lvapcl)=r(1,j,k,lvapcl)+r(1,j,k,lsnow)
           r(1,j,k,lsnow)=0.
           r(1,j,k,lwvap)=r(1,j,k,lvapcl)
         end if
         if(r(1,j,k,lcloud).le.0..and.r(1,j,k,lrain).lt.1.e-5)
     +   then     !Evaporate residual rain if air subsaturated.
           r(1,j,k,lvapcl)=r(1,j,k,lvapcl)+r(1,j,k,lrain)
           r(1,j,k,lrain)=0.
           r(1,j,k,lwvap)=r(1,j,k,lvapcl)
       end if
10     continue
       return
       end
```

Table 6.13b: Listing of Subroutine `cleanup` for Case Example B

```
       subroutine cleanup
c
c      Class I Subroutine called from program pluvius
c
c     Removes nonzero fragments of dependent variables for conditions where
c      particular media have attenuated to negligible magnitudes or do not
c      exist at all.
c
       include 'pluvius.com'
       do i=1,itot
        do j=1,jtot
         do k=1,ktot
          if(r(i,j,k,lcloud).le.O.) then  !Convert cloudborne SO4 to aerosol
           r(i,j,k,lso4g)=r(i,j,k,lso4gcl)
          end if
          if(r(i,j,k,lsnow).le.O.) then
c
c      If no snow, then transfer snow-borne species to rain medium.
c
           r(i,j,k,lso2rain)=r(i,j,k,lso2rain)+r(i,j,k,lso2snow)
           r(i,j,k,lso2snow)=O.
           r(i,j,k,lso4rain)=r(i,j,k,lso4rain)+r(i,j,k,lso4snow)
           r(i,j,k,lso4snow)=O.
           r(i,j,k,lno3rain)=r(i,j,k,lno3snow)+r(i,j,k,lno3rain)
           r(i,j,k,lno3snow)=O.
           r(i,j,k,lh2o2rain)=r(i,j,k,lh2o2rain)+r(i,j,k,lh2o2snow)
           r(i,j,k,lh2o2snow)=O.
          end if
          if(r(i,j,k,lrain).le.O.) then
c
c      if no rain, then transfer rain-borne species to air medium.
c
           r(i,j,k,lso2gcl)=r(i,j,k,lso2rain)+r(i,j,k,lso2gcl)
           r(i,j,k,lso2rain)=O.
           r(i,j,k,lso4g)=r(i,j,k,lso4g)+r(i,j,k,lso4rain)
           r(i,j,k,lso4rain)=O.
           if(r(i,j,k,lcloud).le.O.) r(i,j,k,lso4gcl)=
     +      r(i,j,k,lso4g)
           r(i,j,k,lno3gcl)=r(i,j,k,lno3gcl)+r(i,j,k,lno3rain)
           r(i,j,k,lno3rain)=O.
           r(i,j,k,lh2o2gcl)=r(i,j,k,lh2o2gcl)+r(i,j,k,lh2o2rain)
           r(i,j,k,lh2o2rain)=O.
          end if
```

Table 6.13b, Continued

```
          end do
         end do
        end do
        return
        end
```

Table 6.14: Listing of Subroutine **`collect`**

```
        subroutine collect
c
c       Class II subroutine called from program pluvius
c
        include 'pluvius.com'
        data iend /-7/
c
c       Routine does output to disk file for post-processing.
c       Parameters are output at time intervals = dtdisk.
c
        if (itdisk .eq. O) return
c
c      Initial entry - write "heading" info. to file.
c
        if (iend .eq. -7) then
          open( unit=21, file='diskbkup', status='unknown',
     +    form='unformatted')
          rewind 21
          iend = O
          tdisk = tdisk - .O1*dt
          ntotp1 = ltot2 + 1
          name(ntotp1) = 'cair'
          write(21) mtitle
          write(21) itot, jtot, ktot, ntotp1
          write(21) (name(k), k=1,ntotp1)
          write(21) deltax(1), deltay(1), deltas(1)
        end if
        write(21) t,
```

Table 6.14, Continued

```
     + ((((r(i,j,k,l), i=1,itot), j=1,jtot), k=1,ktot),
     + l=1,ltot2),
     + (((cair(i,j,k), i=1,itot), j=1,jtot), k=1,ktot)
       write(20,3001) t
       tdisk=tdisk+dtdisk
c
c      When itdisk=2, save only one time frame.  By setting iend=-7,
c      subsequent entry will overwrite the previous data.
       if (itdisk.eq.2) then
         close(21)
         iend=-7
       end if
 3001  format(1h ,'writing to disk occurred at t = ',e10.3)
       return
       end
```

Table 6.15: Listing of Subroutine **restart**

```
       subroutine restart
c
c      Class II subroutine called from subroutine inputgn
c
c      Initializes the dependent variables by reading their values from the
c      backup file written by a previous run.  That backup file must
c      be renamed from "diskbkup" to "rstinpt".  Logical unit 21 is used
c      for reading the file.
c
c      trestart - the record (or time slice) on the backup file having this
c      time is used.
c
       include 'pluvius.com'

       character*80 mtitlerst
       character*10 namerst(nnammx)
c
c      Open "rstinpt" file.
c
```

Table 6.15, Continued

```
      open( unit=21, file='rstinpt', status='old',
     +           form='unformatted', iostat=ios )
      if (ios .ne. 0) then
          write(*,9010) ios
          stop
      end if
9010  format( / '*** Error in subroutine restart',
     +    ' while attempting to open file rstinpt' /
     +    'iostat value =', i10 / )

c
c     Read "header" information and compare with current simulation values.
c
      read(21) mtitlerst
      read(21) irst, jrst, krst, nrst
      l2rst = nrst - 1

c     Check l2rst first to avoid possibility of read error if nrst > nnammx.
      if (l2rst .ne. ltot2) then
          write(*,9020) l2rst
          stop
      end if
9020  format( / '*** Error in subroutine restart',
     +    ' -- run ltot2 and restart file ltot2 differ' /
     +    'restart file ltot2=', i10 / )

      read(21) (namerst(n), n=1,nrst)
      read(21) deltaxrst, deltayrst, deltasrst

      if ( (irst .ne. itot) .or. (deltaxrst .ne. deltax(1)) .or.
     +     (jrst .ne. jtot) .or. (deltayrst .ne. deltay(1)) .or.
     +     (krst .ne. ktot) .or. (deltasrst .ne. deltas(1)) ) then
          write(*,9030) irst, jrst, krst,
     +           deltaxrst, deltayrst, deltasrst
          stop
      end if
9030  format( / '*** Error in subroutine restart',
     +    ' -- run grid and restart file grid differ' /
     +    'restart file i/j/ktot =', 3i10 /
     +    'restart file deltax/y/s =', 3(1pe15.7) / )
      do l = 1, ltot2
          if (name(l) .ne. namerst(l))
```

Table 6.15, Continued

```
     +           write(*,9040) l, name(l), namerst(l)
       end do
9040   format( '*** subroutine restart -- names differ',
     +     ' -- l, name, namerst = ', i5, 2(3x, a) )
c
c      Read time slices until slice with time = trestart is found.
c
100    read(21,end=200) t,
     + ((((r(i,j,k,l), i=1,itot), j=1,jtot), k=1,ktot), l=1,ltot2),
     + (((cair(i,j,k), i=1,itot), j=1,jtot), k=1,ktot)

       if (abs(t-trestart) .le. 0.1*dt) then
           close( unit=21 )
           return
       else if (t .lt. trestart) then
           goto 100
       else
           write(*,9050) t
           stop
       end if
9050   format( / '*** Error in subroutine restart',
     +     ' -- trestart not found' /
     +     'last time read from restart file = ', f20.5 / )
200    write(*,9060)
       stop
9060   format( / '*** Error in subroutine restart',
     +     ' -- end of file without finding trestart' / )
       end
```

Table 6.16: Listing of Subroutine **cloudinit**

```
       subroutine cloudinit
c
c      Class II subroutine called from subroutine init
c
```

Table 6.16, Continued

```
c      Initializes fields of cloud water, rain, snow, (vapor + cloud water),
c      and temperature by reading them from the backup file written by
c      the Case Example A run.
c
c      The backup file from the case example a run must be renamed from
c      "diskbkup" to "diskbkup.exa".
c      The first record in the file (= the first time slice) is used.
c      Logical unit 21 is used for reading the file.
c
       include 'pluvius.com'
       character*80 mtitledum
       character*10 namedum
       open( unit=21, file='diskbkup.exa', status='old',
     +            form='unformatted', iostat=ios )
       if (ios .ne. 0) then
           write(*,9010) ios
           stop
       end if
9010   format( / '*** Error in subroutine cloudinit',
     +     ' while attempting to open file diskbkup.exa' /
     +     'iostat value =', i10 / )
       read(21) mtitledum
       read(21) idum, jdum, kdum, ndum
       read(21) (namedum, n=1,ndum)
       read(21) deltaxdum, deltaydum, deltasdum
       if ( (idum .ne. itot) .or. (deltaxdum .ne. deltax(1)) .or.
     +      (jdum .ne. jtot) .or. (deltaydum .ne. deltay(1)) .or.
     +      (kdum .ne. ktot) .or. (deltasdum .ne. deltas(1)) ) then
           write(*,9020) idum, jdum, kdum,
     +            deltaxdum, deltaydum, deltasdum
           stop
       end if
9020   format( / '*** Error in subroutine cloudinit',
     +     ' -- run grid and disk file grid differ' /
     +     'disk file i/j/ktot =', 3i10 /
     +     'disk file deltax/y/s =', 3(1pe15.7) / )

       if (ndum .ne. 8) then
           write(*,9030) ndum
           stop
       end if
9030   format( / '*** Error in subroutine cloudinit',
```

Table 6.16, Continued

```
     +    ' -- disk file has unexpected number of species =', i10 / )

      read(21) tdum,
     +    (((r(i,j,k,lvapcl), i=1,itot), j=1,jtot), k=1,ktot),
     +    (((r(i,j,k,lrain ), i=1,itot), j=1,jtot), k=1,ktot),
     +    (((r(i,j,k,lsnowl), i=1,itot), j=1,jtot), k=1,ktot),
     +    (((rdum,            i=1,itot), j=1,jtot), k=1,ktot),
     +    (((r(i,j,k,lcloud), i=1,itot), j=1,jtot), k=1,ktot),
     +    (((rdum,            i=1,itot), j=1,jtot), k=1,ktot),
     +    (((r(i,j,k,ltemp ), i=1,itot), j=1,jtot), k=1,ktot),
     +    (((cair(i,j,k),     i=1,itot), j=1,jtot), k=1,ktot)

      close( unit=21 )

      return
      end
```

Table 6.17: Listing of Subroutine **inflowinit**

```
      subroutine inflowinit
c
c     Class I Subroutine called from subroutine init
c
c     Sets values for mixing ratios (and theta-c) at inflow boundaries
c
c     In this version:
c      The inflow values are set equal to the initial mixing ratios at the
c      grid point nearest the boundary.
c      The inflow values are constant in time.  If time-varying values are
c      used, then a different subroutine should be created to calculate (or
c      read) the time varying values at appropriate time intervals and load
c      them into the rinflow--- arrays
c
      include 'pluvius.com'
      if (mtranx .gt. 0) then
          do 100 l = 1, ltot
          do 100 k = 1, ktot
```

Table 6.17, Continued

```
          do 100 j = 1, jtot
                 rinflowlox(j,k,l) = r(1   ,j,k,l)
                 rinflowhix(j,k,l) = r(itot,j,k,l)
100       continue
      end if
      if (mtrany .gt. 0) then
          do 200 l = 1, ltot
          do 200 k = 1, ktot
          do 200 i = 1, itot
                 rinflowloy(i,k,l) = r(i,1   ,k,l)
                 rinflowhiy(i,k,l) = r(i,jtot,k,l)
200       continue
      end if
      if (mtrans .gt. 0) then
          do 300 l = 1, ltot
          do 300 j = 1, jtot
          do 300 i = 1, itot
                 rinflowlos(i,j,l) = r(i,j,1   ,l)
                 rinflowhis(i,j,l) = r(i,j,ktot,l)
300       continue
      end if
      return
      end
```

Table 6.18: Listing of Subroutine `core00`

```
       subroutine core00
c
c      Class II subroutine called from subroutine core
c
c      Performs some calculations which are only required on very first
c      call of core in the event that properties computed in the propsx
c      and/or propsy subroutines are time-invariant.  Since vertical
c      transport properties from propss are computed and stored
c      for one vertical column at a time, there is no mfreqps=0 option.
```

Table 6.18, Continued

```
c
       include 'pluvius.com'
       if ((itot .gt. 1) .and. (mtranx .gt. 0)) then
         if (mfreqpx .eq. 0) call propsx(t)
       end if
       if ((jtot .gt. 1) .and. (mtrany .gt. 0)) then
         if (mfreqpy .eq. 0) call propsy(t)
       end if
       return
       end
```

Table 6.19: Listing of Subroutine **grdxyz**

```
       subroutine grdxyz( dxunif, dyunif, dsunif )
c
c      Class II subroutine called from subroutine inptgn
c
c     grdxyz sets arrays swidth, ywidth, and xwidth.  These are the distances
c     between centers of adjacent grid points, and are used in the filters.
c     If the grids are uniform, the arrays scord, ycord and xcord are also set
c      here.  Otherwise they are read in in subroutine inptgn.
c
       include 'pluvius.com'
       if (dxunif .ne. 0.0) then
         do 10 i = 1, itot
           xcord(i) = xcord(1) + (i-1)*dxunif
10       continue
         munifx = 1
       else
         munifx = 0
       end if
       do 20 i = 1, itot
         ip1 = min0( i+1, itot )
         im1 = max0( i-1, 1 )
         xwidth(i) = abs( 0.5*( xcord(ip1) - xcord(im1) ) )
```

Table 6.19, Continued

```
20     continue
       if (itot .le. 1) xwidth(1) = 1.0
       if (dyunif .ne. 0.0) then
         do 30 j = 1, jtot
              ycord(j) = ycord(1) + (j-1)*dyunif
30       continue
         munify = 1
       else
         munify = 0
       end if
       do 40 j = 1, jtot
         jp1 = min0( j+1, jtot )
         jm1 = max0( j-1, 1 )
         ywidth(j) = abs( 0.5*( ycord(jp1) - ycord(jm1) ) )
40     continue
       if (jtot .le. 1) ywidth(1) = 1.0
       if (dsunif .ne. 0.0) then
         do 50 k = 1, ktot
              scord(k) = scord(1) + (k-1)*dsunif
50         continue
         munifs = 1
       else
         munifs = 0
       end if
       do 60 k = 1, ktot
         kp1 = min0( k+1, ktot )
         km1 = max0( k-1, 1 )
         swidth(k) = abs( 0.5*( scord(kp1) - scord(km1) ) )
60     continue
       if (ktot .le. 1) swidth(1) = 1.0
       return
       end
```

Table 6.20: Listing of Subroutine **sinteg**

```
      subroutine sinteg( minteg, isubt, i, j )
c
c     Class II subroutine called from subroutine core
c
c     Subroutine performs a single integration in the s direction for all
c      species but at specified i, j.
c
c     Subroutine parameters:
c     minteg (= 1 or 2) - specifies that this is the first or second
c     integration in the 2*dt integration cycle.
c     isubt - specifies which subintegration core is in.
c     (minteg and isubt are used in conjunction with control parameters
c     like mfreqps to determine if physical properties should be
c     re-calculated).
c     i, j - the i and j grid points at which the integration is being
c     performed.
c
      include 'pluvius.com'
c
c     First set counters for filtering subroutines.
c
      call sfilter( 0, 0, 0, 0 )
      beta = -2./dtcur
      newcof = 0
      if (mfreqps .eq. 0) then
        write(6,*) 'stop in sinteg: illegal mfreqps', mfreqps
        pause
        stop
      else if (mfreqps .eq. 1) then
        if ((minteg.eq.1) .and. (isubt.eq.1)) newcof = 1
      else if (mfreqps .eq. 10) then
        if (minteg .eq. 1) newcof = 1
      else if (mfreqps.eq.20) then
        newcof = 1
      else
        write(6,*) 'stop in sinteg: illegal mfreqps', mfreqps
        pause
        stop
      end if
c
c     Compute aas and bbs coeffients for governing equations.
c
      if (newcof .eq. 1) call propss(t,i,j)
```

Table 6.20, Continued

```
c
c      Compute  aas,  bbs,  and  ccs  coefficients  for  minimization  integral
c      (arrays share common names with governing-equation coefficients).
c
       call coefss(newcof, beta, i, j)
       tbnd=t+0.5*dtcur
c
c      Set boundary conditions.
c
       call bounds(tbnd,i,j,minteg,isubt)
       do 40 l = 1,ltot
c
c      Load vectors for tridagonal matrix (a, b, c, and d).
c
         call loads(nlo,nhi,i,j,l)
c
c      Solve tridagonal matrix for array v.
c
         call tridag(nlo,nhi,a,b,c,d,v)
c
c      Filter array v.
c
         call sfilter(1,i,j,l)
         do 30 k = nlo,nhi
           rclm(k,l) = v(k)
 30      continue
 40    continue
       return
       end
```

Table 6.21: Listing of Subroutine **propss**

```
       subroutine propss(tdum,i,j)
c
c      Class II subroutine called from subroutine sinteg
c
       include 'pluvius.com'
c
c      subroutine  *****  propss  *****
c
c     Supplies "physical properties" in the form of coefficients aas, bbs, and
c      ccs for the s-components of the governing equations. One interrogation
c      returns properties corresponding to time tdum for all species and all
c      vertical points above horizontal position i,j for species equations:
c
c         aas = kzz  (generated in subroutine transport)
c         bbs = - wair  - wfall  + (kzz/cair) * d(cair)/dz
c         ccs = -(1./cair) * d(wfall*cair)/dz
c
c     Transport computes vertical transport parameters: fall velocities [in
c      vertcl(i,j,k,lprops(l))], composite' diffusion coefficients (natural +
c      differential spread of falling hydrometeors) [in aas(k,lprops(l))]
c
       call transport(i,j)
c
c      Now compute bbs(k,lprops(l)) array.
c
       ds2inv = 0.5/deltas(1)
       do 10 ll = 1,lpstot
         bbs(1,ll) = aas(1,ll)*(cairclm(2) - cairclm(1))
     +   /(deltas(1)*cairclm(1))
     +   -windsclm(1)-vertcclm(1,ll)
         bbs(ktot,ll) = aas(ktot-1,ll)*(cairclm(ktot)-cairclm(ktot-1))
     +   /(deltas(ktot-1)*cairclm(ktot))
     +   -windsclm(ktot)-vertcclm(ktot,ll)
         ccs(1,ll) =  -(cairclm(2)*vertcclm(2,ll)-
     +   cairclm(1)*vertcclm(1,ll))/(deltas(1)*cairclm(1))
         ccs(ktot,ll) =  -(cairclm(ktot)*vertcclm(ktot,ll)-
     +   cairclm(ktot-1)*vertcclm(ktot-1,ll))/
     +   (deltas(ktot-1)*cairclm(ktot))
         if (munifs .gt. 0) then
c
c      case for uniform grid:
```

Table 6.21, Continued

```
c
           do 20 k = 2,ktot-1
             bbs(k,ll) = - windsclm(k) - vertcclm(k,ll)
     +       + 0.5*(aas(k,ll) + aas(k-1,ll))*
     +         (cairclm(k+1) - cairclm(k-1))*ds2inv/cairclm(k)
             ccs(k,ll) =  -(cairclm(k+1)*vertcclm(k+1,ll)-
     +       cairclm(k-1)*vertcclm(k-1,ll))*ds2inv/cairclm(k)
 20        continue
         else
c
c      case for non-uniform grid:
c
           do 30 k = 2, ktot-1
             bbs(k,ll) = - windsclm(k) - vertcclm(k,ll)
     +       + 0.5*(aas(k,ll) + aas(k-1,ll))*
     +         (fdcofs(1,k)*cairclm(k-1)
     +       +  fdcofs(2,k)*cairclm(k  )
     +       +  fdcofs(3,k)*cairclm(k+1))/cairclm(k)
             ccs(k,ll) =
     +       - (fdcofs(1,k)*cairclm(k-1)*vertcclm(k-1,ll)
     +       +  fdcofs(2,k)*cairclm(k  )*vertcclm(k  ,ll)
     +       +  fdcofs(3,k)*cairclm(k+1)*vertcclm(k+1,ll))
     +        /cairclm(k)
 30         continue
         end if
 10    continue
c
c      Save values at boundaries (k = 1 and ktot) for possible use with
c      outflow b. c.'s:
c
       do 40 ll = 1,lpstot
         asoflo(ll) = 0.
         bsoflo(ll) = -windsclm(1) - vertcclm(1,ll)
         csoflo(ll) = ccs(1,ll)
         asofhi(ll) = 0.
         bsofhi(ll) = -windsclm(ktot) - vertcclm(ktot,ll)
         csofhi(ll) = ccs(ktot,ll)
40     continue
       return
       end
```

Table 6.22: Listing of Subroutine **coefss**

```
      subroutine coefss( newcof, beta, i, j )
c
c      Class II subroutine called from subroutine sinteg
c
c      Computes coefficients of the terms in the minimization integral
c       aas(k,lprops(l)), bbs(k,lprops(l)), and ccs(k,lprops(l)).
c
c                 beta  - pertains to -2./dt.
c                 newcof - if 1, coefficients must be fully recalculated
c                 if 0, coefficients can be obtained from previous values
c
      include 'pluvius.com'
      data init / 1 /
      if (newcof.ne.1.and.init.ne.1) then
c
c     If the physical transport parameters have not changed, then the min.
c      integral coefficients can be obtained from the previous values.
c
        factor = -betold+beta
        if(abs(factor).le.abs(beta*1.e-6)) return
        do 10 ll = 1,lpstot
          do 10 k = 1,ktot
            aas(k,ll) = aas(k,ll)+factor*ct3b(3,k)
            bbs(k,ll) = bbs(k,ll)+factor*ct3df(3,k)
            ccs(k,ll) = ccs(k,ll)+factor*ct3h(3,k)
 10     continue
        betold = beta
        return
      else
c
c     Otherwise, the min. integral coefficients must be completely recomputed.
c
        do 20 ll = 1,lpstot
          do 30 k = 1,ktot
            a(k) = aas(k,ll)
            b(k) = bbs(k,ll)
            c(k) = ccs(k,ll)
 30       continue
c
c      interior grid points:
c
          do 40 k = 2,ktot-1
          aas(k,ll) = ct1a(3,k)*a(k-1)-
```

Table 6.22, Continued

```
     +       .5*b(k-1)+ct3b(3,k)*(c(k-1)+beta)
     +       -ct2a*(b(k)-b(k-1))
     +       +ct3a(3,k)*(c(k)-c(k-1))
             bbs(k,ll) = -ct1a(3,k)*a(k-1)-
     +       ct1c(3,k)*a(k)+
     +       ct3d(3,k)*(beta+c(k-1))+
     +       ct3f(3,k)*(beta+c(k))
     +       -ct2c*(b(k+1)-b(k-1))
     +       +ct3e(3,k)*(c(k+1)-c(k))
     +       +ct3c(3,k)*(c(k)-c(k-1))
             ccs(k,ll) = +ct1c(3,k)*a(k)
     +       +.5*b(k)+ct3h(3,k)*(c(k)+beta)
     +       +ct2c*(b(k+1)-b(k))
     +       +ct3e(3,k)*(c(k+1)-c(k))
 40        continue
           aas(1,ll) = 0.0
           bbs(1,ll) = 0.0
           ccs(1,ll) = 0.0
           aas(ktot,ll) = 0.0
           bbs(ktot,ll) = 0.0
           ccs(ktot,ll) = 0.0
c
c                 Fill border values for flux b.c.'s:
c
           if (ibclos(ll) .eq. 1) then
             bbs(1,ll) = -ct1c(3,1)*a(1)-ct2c*(b(2)
     +       -b(1))-.5*b(1)+ct3e(3,1)*(c(2)-c(1))+
     +       ct3f(3,1)*(c(1)+beta)
             ccs(1,ll) = ct1c(3,1)*a(1)+ct2c*(b(2)
     +       -b(1))+.5*b(1)+ct3e(3,1)*(c(2)-c(1))+
     +       ct3h(3,1)*(c(1)+beta)
           end if
           if (ibchis(ll) .eq. 1) then
             aas(ktot,ll) = ct1a(3,ktot)*a(ktot-1)
     +       -ct2a*(b(ktot)-b(ktot-1))-.5*b(ktot-1)
     +       +ct3a(3,ktot)*(c(ktot)-c(ktot-1))+
     +       ct3b(3,ktot)*(c(ktot-1)+beta)
             bbs(ktot,ll) = -ct1a(3,ktot)*a(ktot-1)
     +       +ct2a*(b(ktot)-b(ktot-1))+.5*b(ktot-1)
     +       +ct3c(3,ktot)*(c(ktot)-c(ktot-1))+
     +       ct3d(3,ktot)*(c(ktot-1)+beta)
           end if
```

Table 6.22, Continued

```
  20        continue
          init = 0
          betold = beta
        end if
        return
        end
```

Table 6.23: Listing of Subroutine **bounds**

```
       subroutine bounds(tdum,i,j)
c
c      Class I subroutine called from subroutine sinteg
c
c      Computes s-boundary "fluxes" (interpreted here as velocity times mixing
c       ratio and thus having units of l/t).
c
       include 'pluvius.com'
c
c      For Case Examples A & B the vertical wind component is zero at the top
c      and bottom boundaries, thus top and bottom fluxes are set to zero at
c      this point.  Transport of precipitation across the bottom boundary is
c      handled automatically within the numerical integration scheme as an
c      "outflow" condition.
c
       data ifirst / 1 /
       do 1000 l = 1, ltot
           flxlsclm(l) = 0.
           flxhsclm(l) = 0.
1000   continue
       return
       end
```

Table 6.24: Listing of Subroutine **`loads`**

```
       subroutine loads(nlo,nhi,i,j,l)
c
c      Class II subroutine called from subroutine sinteg
c
c      Loads elements of tridiagonal matrix for solutions in the s-dimension.
c      first consolodates coefficients of the integral equations (aas, bbs,
c       ccs) to obtain internal elements.  Then computes boundary conditions
c      to obtain external elements.
c
c      Subroutine is called once for each s vector in the domain's matrix.
c
       include 'pluvius.com'

       nlo = 1
       nhi = ktot
       ll = lprops(l)
       do 10 k = 1,ktot
         a(k) = aas(k,ll)
         b(k) = bbs(k,ll)
         c(k) = ccs(k,ll)
 10    continue
c
c      compute d array at interior grids
c
       ktotm1 = ktot - 1
       factor = 4./dtcur
       do 1200 k = 2, ktotm1
         d(k) = - (a(k) + factor*ct3b(3,k))*rclm(k-1,l)
     +   - (b(k) + factor*ct3df(3,k))*rclm(k,l)
     +   - (c(k) + factor*ct3h(3,k))*rclm(k+1,l)
1200   continue
c
c      Compute d at k=1 boundary, and adjust coefs. for boundary conditions:
c
c      Outflow exists when (windsclm+vertcclm) and deltas have opposite sign
c      (their product is negative);
c      otherwise inflow or no-flow, in which case
c     ibclos(ll) = 1 --> flux bc
c     ibclos(ll) = other --> forced outflow which is treated as zero flux.
c
       k = 1
       dumwinds = -bsoflo(ll)
```

Table 6.24, Continued

```
       if (dumwinds*deltas(1) .lt. O.O) then
c      ( outflow case )
c
         c(1) = bsoflo(ll)/deltas(1)
         b(1) = -c(1) -1./dtcur + csoflo(ll)
         a(1) = O.
         d(1) = -rclm(1,l)/dtcur
       else
c
c       ( no outflow )
c
         b(1) = b(1) + bsoflo(ll)
         d(k) = -(b(k) + factor*ct3df(3,k))*rclm(k,l)
     +   -(c(k) + factor*ct3h(3,k))*rclm(k+1,l)
         if (ibclos(ll) .eq. 1) d(1) = d(1) - 2.*flxlsclm(l)
       end if
c
c      Compute d at k=ktot boundary, & adjust coefs. for boundary conditions.
c
c      Outflow exists when (windsclm+vertcclm) deltas have the same sign
c     (their product is positive);
c     otherwise inflow or no-flow, in which case
c     ibchis(ll) = 1 --> flux bc
c     ibchis(ll) = other --> forced outflow which is treated as zero flux.
c
       k = ktot
       dumwinds = -bsofhi(ll)
       if (dumwinds*deltas(1) .gt. O.O) then
c
c      ( outflow case )
c
         a(ktot) = -bsofhi(ll)/deltas(ktot-1)
         b(ktot) = -a(ktot) - 1./dtcur + csofhi(ll)
         c(ktot) = O.
         d(ktot) = -rclm(ktot,l)/dtcur
       else
c
c      ( no outflow )
c
         b(ktot) = b(ktot) - bsofhi(ll)
         d(k) = - (a(k) + factor*ct3b(3,k))*rclm(k-1,l)
     +   - (b(k) + factor*ct3df(3,k))*rclm(k,l)
```

Table 6.24, Continued

```
          if (ibchis(ll) .eq. 1) d(ktot) = d(ktot) + 2.*flxhsclm(l)
        end if
        return
        end
```

Table 6.25: Listing of Subroutine **transport**

```
        subroutine transport(i,j)
c
c       Class I subroutine called from subroutines matbal and propss
c
c      Transport computes fall velocities (negative downward) and spread
c      parameters of rain and snow in cm/sec and cm**2/sec units.  Variable
c      names are vertcclm(k,lprops(lrain)) and vertcclm(k,lprops(lsnow)),
c      where k is the vertical grid index and lprops denote the transport
c      regimes.
c
c
        include 'pluvius.com'
        dimension rain(kmaxd),snow(kmaxd),precip(kmaxd)
        data istart /1/
c
c       Initialize arrays:
c
        if(istart.eq.1) then
          istart=O
          do 1O k=1,ktot
           do 1O l=1,lpsmax
             vertcclm(k,l)=O.
 1O       continue
        end if
c
c       Define working concentrations in gram-moles per cubic meter.
c
```

Table 6.25, Continued

```
      do 20 k=1,ktot
        rain(k)=rclm(k,lrain)*cairclm(k)*1.0e6
        snow(k)=rclm(k,lsnow)*cairclm(k)*1.0e6
        rain(k)=amax1(0.,rain(k))
        snow(k)=amax1(0.,snow(k))
        precip(k)=rain(k)+snow(k)
 20   continue
c
c     Now compute transport properties.
c
      do 30 k=1,ktot
        drain(k)=3.53*dtcur*rain(k)**.25
        dsnow(k)=0.
c
c     drain, dsnow = estimated effective diffusivities of rain & snow arising
c     from differential settling velocities, m2/s.
c
        vrain(k)=-0.01
        vsnow(k)=-0.01
c
        if(precip(k).gt.1.e-5) then
c
c     Compute rain fall velocities in m/s based on Marshall-Palmer
c     distribution.  rlambd is lambda in M-P distribution.
c
          rlambd=1149.*precip(k)**(-.25)
          aol=1.02e-3*rlambd
          eaol=exp(-aol)
          fact3=1.+aol*(1.+(aol/2.)*(1.+aol/3.))
          fact4=1.-eaol*(fact3+(aol**4)/24.)
          fact3=1.-eaol*fact3
          fact34=sqrt(fact3*fact4)
          vrain(k)=-(1.-fact34)*252.2/sqrt(rlambd)-
     +    16220.*fact4/rlambd
        end if
c
        if(snow(k).gt.1.e-5) then
c
c     Compute snow fall velocities in m/s based on a Marshall-Palmer
c     distribution.  slambd is lambda in M-P distribution.
c
          tc=rclm(k,ltemp)-273.15
```

Table 6.25, Continued

```
           y0flak=5.681e6*exp(-0.0878*amin1(tc,0.))
           slambd=2.148*(y0flak/precip(k))**0.3333
           aflake=1./slambd
           vsnow(k)=-4.06*aflake**.206
           vbarg=.00156*aflake**.206
           if(aflake.lt.3.37e-4) then
             vsnow(k)=-1296.*aflake**.927
           end if
c
c      Calculate estimated effective snow diffusivity, m2/s.
c
           if(aflake.lt.4.6e-5) vbarg=6.95*aflake**.927
           dsnow(k)=dtcur*vbarg*vbarg
         end if
         if(precip(k).gt.0..and.rain(k).gt.0.)
     +   vrain(k)=
     +   (vrain(k)*rain(k) +
     +    vsnow(k)*snow(k))/precip(k)
 30    continue
c
c                 convert to cm units
c
        do 40 k=1,ktot
         vrain(k)=vrain(k)*100.
         vsnow(k)=vsnow(k)*100.
         drain(k)=difusclm(k,2)+drain(k)*1.0e4
         dsnow(k)=difusclm(k,2)+dsnow(k)*1.0e4
 40     continue
c
c      Do vertical smoothing of rain and snow fallspeeds and diffusivities.
c
       do 50 k=1,ktot-2
         vrain(k)=(vrain(k)+vrain(k+1)+vrain(k+2))/3.
         vsnow(k)=(vsnow(k)+vsnow(k+1)+vsnow(k+2))/3.
         drain(k)=(drain(k)+drain(k+1)+drain(k+2))/3.
         dsnow(k)=(dsnow(k)+dsnow(k+1)+dsnow(k+2))/3.
 50    continue
c
c      Load vertcclm and aas arrays
c
       do 60 k=1,ktot
         vertcclm(k,lprops(lrain))=vrain(k)
```

Table 6.25, Continued

```
          vertcclm(k,lprops(lsnow))=vsnow(k)
          aas(k,1)=difusclm(k,1)
          aas(k,2)=difusclm(k,2)
          aas(k,lprops(lrain))=drain(k)
          aas(k,lprops(lsnow))=dsnow(k)
 60     continue
        return
        end
```

Table 6.26: Listing of Subroutine **xinteg**

```
        subroutine xinteg( minteg )
        include 'pluvius.com'
c
c       Class II subroutine called from subroutine core
c
c       Subroutine performs a single integration in the x direction.
c
c       minteg (=1 or 2) specifies which integration in the x-y-z-chem-z-y-x
c       cycle this is.
c
c       First, set counters for filtering subroutines
c
        call xfilter( 0, 0, 0, 0 )
        beta = -2./dt
        newcof = 0
        if (mfreqpx .eq. 0) then
          continue
        else if (mfreqpx .eq. 1) then
          if (minteg.eq.1) newcof=1
        else if(mfreqpx.eq.2) then
          newcof=1
```

Table 6.26, Continued

```
       end if
c
c      Compute aax and bbx coeffients for governing equations.
c
       if (newcof .eq. 1) call propsx(t)
c
c     Compute aax, bbx, and ccx coefficients for minimization integral (arrays
c      share common names with governing-equation coefficients).
c
       call coefsx(newcof, beta)
c
c      Set boundary conditions.
c
       call boundx(tbound,minteg)
       do 10 l = 1,ltot
         do 10 k = 1,ktot
           do 10 j = 1,jtot
c
c      Load vectors for tridagonal matrix (a, b, c, and d).
c
             call loadx(nlo,nhi,j,k,l)
c
c      Solve tridagonal matrix for array v.
c
             call tridag(nlo,nhi,a,b,c,d,v)
c
c      Filter array v.
c
             call xfilter( 1, j, k, l )
             do 30 i = nlo,nhi
               r(i,j,k,l) = v(i)
 30          continue
 10    continue
       return
       end
```

Table 6.27: Listing of Subroutine **propsx**

```
      subroutine propsx(tdum)
       include 'pluvius.com'
c
c      Class II subroutine called from subroutine xinteg
c
c      Supplies "physical properties" in the form of coefficients aax and bbx
c       for the x-components of the governing equations. One interrogation
c       returns properties corresponding to time tdum for all species and all
c       grid points.
c
c      First load arrays with physical transport properties.
c
       do 10 ll = 1, lpxtot
         do 10 k = 1, ktot
           do 10 j = 1, jtot
             do 10 i = 1, itot
               aax(i,j,k,ll) = diffux(i,j,k)
 10    continue
       dx2inv = 0.5/deltax(1)
       do 20 ll = 1,lpxtot
         do 20 k = 1,ktot
           do 20 j = 1,jtot
             bbx(1,j,k,ll) = -windx(1,j,k)
            bbx(1,j,k,ll)=bbx(1,j,k,ll)+aax(1,j,k,ll)*
     +       (cair(2,j,k)-cair(1,j,k))/(deltax(1)*cair(1,j,k))
             bbx(itot,j,k,ll) = -windx(itot,j,k)
            bbx(itot,j,k,ll)=bbx(itot,j,k,ll)+aax(itot-1,j,k,ll)*
     +       (cair(itot,j,k)-cair(itot-1,j,k))/
     +       (deltax(itot-1)*cair(itot,j,k))
            if (munifx.ne.0) then
c
c     case for uniform grid:
c
               do 30 i = 2,itot-1
                 bbx(i,j,k,ll) = -windx(i,j,k)
                bbx(i,j,k,ll)=bbx(i,j,k,ll)+0.5*(aax(i,j,k,ll)+
     +          aax(i-1,j,k,ll))*(cair(i+1,j,k)-cair(i-1,j,k))*dx2inv/
     +            cair(i,j,k)
 30            continue
             else
c
c     case for a nonuniform grid:
```

Table 6.27, Continued

```
c
             do 40 i=2, itot-1
               bbx(i,j,k,ll) = -windx(i,j,k)
               bbx(i,j,k,ll) = bbx(i,j,k,ll)+
     +         0.5*(aax(i,j,k,ll)+aax(i-1,j,k,ll))*
     +         (fdcofx(1,i)*cair(i-1,j,k)+
     +         fdcofx(2,i)*cair(i,j,k)+
     +         fdcofx(3,i)*cair(i+1,j,k))/cair(i,j,k)
 40          continue
             end if
 20    continue
c
c      Pack arrays for possible use with outflow b.c.'s.
c
       do 50 ll = 1,lpxtot
         do 50 k = 1,ktot
           do 50 j = 1,jtot
             axoflo(j,k,ll) = 0.
             bxoflo(j,k,ll) = -windx(1,j,k)
             axofhi(j,k,ll) = 0.
             bxofhi(j,k,ll) = -windx(itot,j,k)
50     continue
       return
       end
```

Table 6.28: Listing of Subroutine **coefsx**

```
       subroutine coefsx( newcof, beta )
c
c      Class II subroutine called from subroutine xinteg
c
c       Computes coefficients of terms in the minimization integral
c       aax(i,j,k,lpropx(l)), bbx(i,j,k,lpropx(l)), and ccx(i,j,k,lpropx(l)).
c       beta  - pertains to -2./dt.
```

Table 6.28, Continued

```
c      newcof - if 1, coefficients must be fully recalculated
c       if 0, coefficients can be obtained from previous values
c
      include 'pluvius.com'
      data init /1/
      if (newcof .ne. 1.and.init.ne.1) then
c
c      If the physical transport parameters for x have not changed, then the
c      min. integral coefficients can be obtained from the previous values.
c
        factor = -betold+beta
        if(abs(factor).le.abs(beta*1.*1.e-6)) return
        do 10 ll = 1,lpxtot
          do 10 k = 1,ktot
            do 10 j = 1,jtot
              do 10 i = 1,itot
                aax(i,j,k,ll) = aax(i,j,k,ll)+factor*ct3b(1,i)
                bbx(i,j,k,ll) = bbx(i,j,k,ll)+factor*ct3df(1,i)
                ccx(i,j,k,ll) = ccx(i,j,k,ll)+factor*ct3h(1,i)
 10     continue
        betold = beta
        return
c
c      Otherwise, the min. integral coefficients must be completely
c      recomputed.
c
      else
        do 20 ll = 1,lpxtot
          do 20 k = 1,ktot
            do 20 j = 1,jtot
              do 30 i = 1,itot
                a(i) = aax(i,j,k,ll)
                b(i) = bbx(i,j,k,ll)
 30           continue
              do 40 i = 2,itot-1
                aax(i,j,k,ll) = ct1a(1,i)*a(i-1)-
     +          .5*b(i-1)+ct3b(1,i)*beta
     +          -ct2a*(b(i)-b(i-1))
                bbx(i,j,k,ll) = -ct1a(1,i)*a(i-1)-
     +          ct1c(1,i)*a(i)+beta*ct3df(1,i)
     +          -ct2c*(b(i+1)-b(i-1))
                ccx(i,j,k,ll) = ct1c(1,I)*a(I)+
```

Table 6.28, Continued

```
     +          .5*b(i)+ct3h(1,i)*beta
     +          +ct2c*(b(i+1)-b(i))
 40            continue
               aax(1,j,k,ll) = 0.
               bbx(1,j,k,ll) = 0.
               ccx(1,j,k,ll) = 0.
               aax(itot,j,k,ll) = 0.
               bbx(itot,j,k,ll) = 0.
               ccx(itot,j,k,ll) = 0.
c
c      Fill border values for flux b.c.'s.
c
                if (ibclox(ll) .eq. 1) then
                  bbx(1,j,k,ll) = -ct1c(1,1)*a(1)-ct2c*(b(2)
     +            -b(1))-.5*b(1)+
     +            ct3f(1,1)*beta
                  ccx(1,j,k,ll) = ct1c(1,1)*a(1)+ct2c*(b(2)
     +            -b(1))+.5*b(1)+
     +            ct3h(1,1)*beta
                end if
                if (ibchix(ll) .eq. 1) then
                  aax(itot,j,k,ll) = ct1a(1,itot)*a(itot-1)
     +            -ct2a*(b(itot)-b(itot-1))-.5*b(itot-1)
     +            +ct3b(1,itot)*beta
                  bbx(itot,j,k,ll) = -ct1a(1,itot)*a(itot-1)
     +            +ct2a*(b(itot)-b(itot-1))+.5*b(itot-1)
     +            +ct3d(1,itot)*beta
                end if
 20         continue
            init = 0
            betold = beta
            return
          end if
          end
```

Table 6.29: Listing of Subroutine **boundx**

```
      subroutine boundx(tdum)
c
c       Class I subroutine called from subroutine xinteg
c
c       Sets "pseudo inflow fluxes" at x boundaries.
c       tdum = model time (s) for which the fluxes are to be calculated.
c
c       Boundx sets
c       flxlx(j,k,l) = "pseudo inflow flux" for species l across the lower
c           x boundary at 1,j,k
c       flxhx(j,k,l) = "pseudo inflow flux" for species l across the upper
c           x boundary at itot,1,k
c       "pseudo inflow flux" = true inflow flux (moles/cm**2/s)
c           divided by air molar density.  Units = (moles/mole-air)*cm/s.
c           [For theta-c, units = (degrees-K)*cm/s]
c
c       flxlx and flxhx are only used at inflow grids where the x wind
c       component is directed into the model domain.  Values for flxlx and
c       flxhx are calculated for all points along the x boundary, but values
c       at outflow grids are never used.
c
c       In Case Examples A and B, the x-dimension is not used; thus this
c       subroutine is not called for these examples.
c
        include 'pluvius.com'
        data ifirst / 1 /
        if (ifirst .ne. 1) return
        ifirst = -1
        do 100 l = 1, ltot
        do 100 k = 1, ktot
        do 100 j = 1, jtot
            flxlx(j,k,l) = rinflowlox(j,k,l)*windx(j,1   ,k)
            flxhx(j,k,l) = rinflowhix(j,k,l)*windx(j,jtot,k)
100     continue
        return
        end
```

Table 6.30: Listing of Subroutine **loadx**

```
      subroutine loadx(nlo,nhi,j,k,l)
c
c     Class II subroutine called from subroutine xinteg
c
c      Loads elements of tridiagonal matrix for solutions in the x-dimension.
c      First consolodates coefficients of the integral equations (aax, bbx,
c      ccx) to obtain internal elements.  Then computes boundary conditions
c      to obtain external elements.
c
c      Subroutine is called once for each x vector in the domain's matrix.
c
      include 'pluvius.com'
      ll = lpropx(l)
      nlo=1
      nhi=itot
      do 10 i = 1,itot
        a(i) = aax(i,j,k,ll)
        b(i) = bbx(i,j,k,ll)
        c(i) = ccx(i,j,k,ll)
 10   continue
      factor = 4./dtcur
      do 20 i=2,itit-1
        d(i)=(a(i)+factor*ct3b(1,i))*r(i-1,j,k,l)
     +   -(b(i)+factor*ct3df(1,i))*r(i,j,k,l)
     +   -(c(i)+factor*ct3h(1,i))*r(i+1,j,k,l)
 20   continue
c
c      Compute d at i=1 boundary, and adjust coeffs for boundary conditions.
c
c      Outflow exists when windx and deltax have opposite sign (their product
c      is negative).
c      Otherwise inflow (or no flow), in which case
c      ibclox(ll) = 1 ---> flux b.c.
c      ibclox(ll) = other ---> forced outflow, which is treated as zero flux.
c
      i=1
      dumwindx = -bxoflo(j,k,ll)
      if (dumwindx*deltax(1) .lt. 0.0) then
c
c      ( outflow case )
c
        c(1) = bxoflo(j,k,ll)/deltax(1)
```

Table 6.30, Continued

```
          b(1) = -c(1) - 1./dtcur
          a(1) = O.
          d(1) = -r(1,j,k,l)/dtcur
        else
c
c        ( no outflow )
c
          b(1) = b(1) - windx(1,j,k)
          d(i) = - (b(i) + factor*ct3df(1,i))*r(i,j,k,l)
     +    -(c(i) + factor*ct3h(1,i)) *r(i+1,j,k,l)
          if (ibclos(ll) .eq. 1) d(i) = d(i) - 2.*flxlx(j,k,l)
        end if
c
c       Compute d at i=itot boundary, and adjust coefs. for bound. conditions.
c
c       Outflow exists when windx and deltax have the same sign (their product
c       is positive).
c       Otherwise inflow or no-flow, in which case
c       ibchix(ll) = 1 --> flux bc
c       ibchix(ll) = other --> forced outflow which is treated as zero flux.
c
        i = itot
        dumwindx = -bxofhi(j,k,ll)
        if (dumwindx*deltax(1) .gt. O.O) then
c
c        ( outflow case )
c
          a(itot) = -bxofhi(j,k,ll)/deltax(itot-1)
          b(itot) = -a(itot) - 1./dtcur
          c(itot) = O.
          d(itot) = -r(itot,j,k,l)/dtcur
        else
c
c        ( no outflow )
c
          b(itot) = b(itot) + windx(itot,j,k)
          d(i) = - (a(i) + factor*ct3b(1,i))*r(i-1,j,k,l)
     +    -(b(i) + factor*ct3df(1,i))*r(i,j,k,l)
          if (ibchix(ll) .eq. 1) d(i) = d(i) + 2.*flxhx(j,k,l)
        end if
        return
        end
```

Table 6.31: Listing of Subroutine **yinteg**

```
       subroutine yinteg( minteg )
       include 'pluvius.com'
c
c      Class II subroutine called from subroutine core
c
c      Subroutine performs a single integration in the y direction.
c
c       minteg (=1 or 2) - specifies which integration this is in the
c       x-y-z-chem-z-y-x integration cycle.
c
c       First set counters for filtering subroutines.
c
       call yfilter( 0, 0, 0, 0 )
       beta = -2./dt
       newcof=0
       if (mfreqpy .eq. 0) then
         continue
       else if (mfreqpy .eq. 1) then
         if(minteg.eq.1) newcof=1
       else if (mfreqpy.eq.2) then
         newcof = 1
       end if
c
c      Compute aay and bby coeffients for governing equations.
c
       if (newcof .eq. 1) call propsy(t)
c
c       Compute aay, bby, and ccy coefficients for minimization integral
c       (arrays share common names with governing-equation coefficients).
c
       call coefsy(newcof, beta)
       tbnd=t+.5*dt
c
c      Set boundary conditions.
c
       call boundy(tbnd,minteg)
         do 10 l = 1,ltot
           do 10 k = 1,ktot
            do 10 i = 1,itot
c
c      Load vectors for tridagonal matrix (a, b, c, and d).
c
```

Table 6.31, Continued

```
             call loady(nlo,nhi,i,k,l)
c
c      Solve tridagonal matrix for array v.
c
             call tridag(nlo,nhi,a,b,c,d,v)
c
c      Filter array v.
c
             call yfilter( 1, i, k, l )
             do 20 j = nlo,nhi
               r(i,j,k,l) = v(j)
 20          continue
 10        continue
        return
       end
```

Table 6.32: Listing of Subroutine **propsy**

```
       subroutine propsy(tdum)
c
c       Class II subroutine called from subroutine yinteg
c
       include 'pluvius.com'
c
c      subroutine  *****  propsy  *****
c
c       Supplies "physical properties" in the form of coefficients aay and
c       bby for the y-components of the governing equations.
c       One interrogation returns properties corresponding to time tdum for
c       all species and all grid points.
c
c      First load physical properties into bby arrays.
c
       do 10 ll = 1, lpytot
```

Table 6.32, Continued

```
        do 10 k = 1, ktot
          do 10 j = 1, jtot
            do 10 i = 1, itot
              aay(i,j,k,ll) = diffuy(i,j,k)
 10     continue
        dy2inv = 0.5/deltay(1)
        do 20 ll = 1,lpytot
          do 20 k = 1,ktot
            do 20 i = 1,itot
c
c       Set bby at boundaries:
c
             bby(i,1,k,ll) = -windy(i,1,k)
            bby(i,1,k,ll)=bby(i,1,k,ll)+aay(i,1,k,ll)*
     +       (cair(i,2,k)-cair(i,1,k))/(deltay(1)*cair(i,1,k))
             bby(i,jtot,k,ll) = -windy(i,jtot,k)
            bby(i,jtot,k,ll)=bby(i,jtot,k,ll)+aay(i,jtot-1,k,ll)*
     +       (cair(i,jtot,k)-cair(i,jtot-1,k))/
     +      (deltay(jtot-1)*cair(i,jtot,k))
c
c       Now set bby for internal points:
c
            if (munify.gt.0) then
c
c       case for a uniform grid
c
               do 30 j = 2,jtot-1
                 bby(i,j,k,ll) = -windy(i,j,k)
                bby(i,j,k,ll)=bby(i,j,k,ll)+0.5*(aay(i,j,k,ll)+
     +          aay(i,j-1,k,ll))*(cair(i,j+1,k)-cair(i,j-1,k))*
     +          dy2inv/cair(i,j,k)
 30            continue
             else
c
c       case for a nonuniform grid
c
             do 40 j=2, jtot-1
               bby(i,j,k,ll) = -windy(i,j,k)
               bby(i,j,k,ll) = bby(i,j,k,ll)+
     +         0.5*(aay(i,j,k,ll)+aay(i,j-1,k,ll))*
     +         (fdcofy(1,j)*cair(i,j-1,k)+
     +         fdcofy(2,j)*cair(i,j,k)+
```

Table 6.32, Continued

```
     +         fdcofy(3,j)*cair(i,j+1,k))/cair(i,j,k)
 40          continue
             end if
 20    continue
c
c       Pack arrays for possible use with outflow b.c.'s:
c
       do 50 ll = 1,lpytot
         do 50 k = 1,ktot
           do 50 i = 1,itot
             ayoflo(i,k,ll) = 0.
             byoflo(i,k,ll) = -windy(i,1,k)
             ayofhi(i,k,ll) = 0.
             byofhi(i,k,ll) = -windy(i,jtot,k)
 50    continue
       return
       end
```

Table 6.33: Listing of Subroutine **coefsy**

```
       subroutine coefsy( newcof, beta )
c
c       Class II subroutine called from subroutine yinteg
c
c      subroutine  ******  coefsy  ******
c
c      Computes coefficients of the the terms in the minimization integral
c      aay(i,j,k,ldif (l), bby(i,j,k,lpropy(l)), and ccy(i,j,k,lpropy(l)).
c
c      beta  - pertains to -2./dt.
c      newcof - if 1, coefficients must be fully recalculated
c           if 0, coefficients can be obtained from previous values
c
       include 'pluvius.com'
```

Table 6.33, Continued

```
       data init/1/
       if (newcof.ne.1.and.init.ne.1)then
c
c       If the physical transport properties for y have not changed, then the
c       min. integral coefficients can be obtained from the previous values.
c
         factor = -betold+beta
         if(abs(factor).le.abs(beta*1.0e-6)) return
           do 10 ll = 1,lpytot
             do 10 k = 1,ktot
               do 10 j = 1,jtot
                 do 10 i = 1,itot
                   aay(i,j,k,ll) = aay(i,j,k,ll)+factor*ct3b(2,j)
                   bby(i,j,k,ll) = bby(i,j,k,ll)+factor*ct3df(2,j)
                   ccy(i,j,k,ll) = ccy(i,j,k,ll)+ct3h(2,j)*factor
 10        continue
         betold = beta
         return
       end if
c
c       Otherwise, the min. integral coefficients must be completely
c       recomputed.
c
       do 20 ll = 1,lpytot
         do 20 k = 1,ktot
           do 20 i = 1,itot
             do 30 j = 1,jtot
               a(j) = aay(i,j,k,ll)
               b(j) = bby(i,j,k,ll)
 30          continue
           do 40 j = 2,jtot-1
             aay(i,j,k,ll) = ct1a(2,j)*a(j-1)-
     +       .5*b(j-1)+ct3b(2,j)*beta
     +       -ct2a*(b(j)-b(j-1))
             bby(i,j,k,ll) = -ct1a(2,j)*a(j-1)-
     +       ct1c(2,j)*a(j)+beta*ct3df(2,j)
     +       -ct2c*(b(j+1)-b(j-1))
             ccy(i,j,k,ll) = ct1c(2,j)*a(j)+
     +       .5*b(j)+ct3h(2,j)*beta
     +       +ct2c*(b(j+1)-b(j))
 40        continue
           aay(i,1,k,ll) = 0.0
```

Table 6.33, Continued

```
          bby(i,1,k,ll) = 0.0
          ccy(i,1,k,ll) = 0.0
          aay(i,jtot,k,ll) = 0.0
          bby(i,jtot,k,ll) = 0.0
          ccy(i,jtot,k,ll) = 0.0
c
c      Fill border values for flux b.c.'s.
c
          if (ibcloy(ll).eq.1) then
            bby(i,1,k,ll) = -ct1c(2,1)*a(1)-ct2c*(b(2)
     +      -b(1))-.5*b(1)+
     +      ct3f(2,1)*beta
            ccy(i,1,k,ll) = ct1c(2,1)*a(1)+ct2c*(b(2)
     +      -b(1))+.5*b(1)+
     +      ct3h(2,1)*beta
          endif
          if (ibchiy(ll).eq.1) then
            aay(i,jtot,k,ll) = ct1a(2,jtot)*a(jtot-1)
     +      -ct2a*(b(jtot)-b(jtot-1))-.5*b(jtot-1)
     +      +ct3b(2,jtot)*beta
            bby(i,jtot,k,ll) = -ct1a(2,jtot)*a(jtot-1)
     +      +ct2a*(b(jtot)-b(jtot-1))+.5*b(jtot-1)
     +      +ct3d(2,jtot)*beta
          endif
 20   continue
      init = 0
      betold = beta
      return
      end
```

Table 6.34: Listing of Subroutine **boundy**

```
      subroutine boundy( tdum )
c
c     Class I subroutine called from subroutine yinteg
c
c     Sets "pseudo inflow fluxes" at y boundaries.
c     tdum = model time (s) for which the fluxes are to be calculated
c
c     boundy sets:
c     flxly(i,k,l) = "pseudo inflow flux" for species l across the lower
c     y boundary at i,1,k.
c     flxhy(i,k,l) = "pseudo inflow flux" for species l across the upper
c     y boundary at i,jtot,k.
c     "pseudo inflow flux" = true inflow flux (moles/cm**2/s)
c      divided by air molar density.  Units = (moles/mole-air)*cm/s.
c      [For theta-c, units = (degrees-K)*cm/s]
c
c     flxly and flxhy are only used at inflow grids where the y wind
c     component is directed in to the model domain.  Values for flxly and
c     flxhy are calculated for all points along the y boundary, but values
c     at outflow grids are never used.
c
c     In Case Examples A and B, both the winds and the inflow mixing ratios
c     are constant in time, so flxly and flxhy are also constant in time.
c
      include 'pluvius.com'
      data ifirst / 1 /
      if (ifirst .ne. 1) return
      ifirst = -1
      do 100 l = 1, ltot
      do 100 k = 1, ktot
      do 100 i = 1, itot
        flxly(i,k,l) = rinflowloy(i,k,l)*windy(i,1   ,k)
        flxhy(i,k,l) = rinflowhiy(i,k,l)*windy(i,jtot,k)
100   continue
      return
      end
```

Table 6.35: Listing of Subroutine **loady**

```
       subroutine loady(nlo,nhi,i,k,l)
c
c     Class II subroutine called from subroutine yinteg
c
c     Loads elements of tridiagonal matrix for solutions in the y-dimension.
c     First consolodates coefficients of the integral equations (aay, bby,
c     ccy) to obtain internal elements, then computes boundary conditions
c     to obtain external elements.
c
c     Subroutine is called once for each y vector in the domain's matrix.
c
       include 'pluvius.com'
       ll = lpropy(l)
       nlo = 1
       nhi = jtot
       do 1100 j = 1,jtot
         a(j) = aay(i,j,k,ll)
         b(j) = bby(i,j,k,ll)
         c(j) = ccy(i,j,k,ll)
1100   continue
       factor = 4./dtcur
       do 1200 j = 2, jtot-1
         d(j) = - (a(j) + factor*ct3b(2,j)) *r(i,j-1,k,l)
     +   - (b(j) + factor*ct3df(2,j))*r(i,j,k,l)
     +   - (c(j) + factor*ct3h(2,j)) *r(i,j+1,k,l)
1200   continue
c
c     Compute d at j=1 boundary, and adjust coefs. for boundary conditions.
c
c     Outflow exists when windy and deltay have opposite sign (their product
c     is negative); otherwise inflow or no-flow, in which case:
c      ibcloy(ll) = 1 --> flux bc
c      ibcloy(ll) = other --> forced outflow which is treated as zero flux
c
       j = 1
       dumwindy = -byoflo(i,k,ll)
       if (dumwindy*deltay(1) .lt. 0.0) then
c
c      ( outflow case )
c
         c(1) = byoflo(i,k,ll)/deltay(1)
         b(1) = -c(1) - 1./dtcur
```

Table 6.35, Continued

```
          a(1) = O.
          d(1) = -r(i,1,k,l)/dtcur
        else
c
c       ( no outflow )
          b(1) = b(1) - windy(i,1,k)
          d(j) = - (b(j) + factor*ct3df(2,j))*r(i,j,k,l)
      +   - (c(j) + factor*ct3h(2,j)) *r(i,j+1,k,l)
          if (ibcloy(ll) .eq. 1) d(j) = d(j) - 2.*flxly(i,k,l)
        end if
c
c      Compute d at j=jtot boundary, and adjust coefs. for boundary conditions.
c
c      Outflow exists when windy and deltay have the same sign
c        (their product is positive)
c      Otherwise inflow or no-flow, in which case:
c        ibchiy(ll) = 1 --> flux bc
c        ibchiy(ll) = other --> forced outflow which is treated as zero flux
c
        j = jtot
        dumwindy = -byofhi(i,k,ll)
        if (dumwindy*deltay(1) .gt. O.O) then
c
c       ( outflow case )
c
          a(jtot) = -byofhi(i,k,ll)/deltay(jtot-1)
          b(jtot) = -a(jtot) - 1./dtcur
          c(jtot) = O.
          d(jtot) = -r(i,jtot,k,l)/dtcur
        else
c
c       ( no outflow )
c
          b(jtot) = b(jtot) + windy(i,jtot,k)
          d(j) = - (a(j) + factor*ct3b(2,j)) *r(i,j-1,k,l)
      +   - (b(j) + factor*ct3df(2,j))*r(i,j,k,l)
          if (ibchiy(ll) .eq. 1) d(j) = d(j) + 2.*flxhy(i,k,l)
        end if
        return
        end
```

Table 6.36: Listing of Subroutine **sfilter**

```
       subroutine sfilter( mfilt, i, j, l )
c
c      Class II subroutine called from subroutines core and sinteg
c
c      Interfaces to foresfilt and bndfilt for s-dimension.
c
       include 'pluvius.com'
       dimension swght(kmaxd)
       data kfilts07 / 7 /
       data kfilts, kfiltssv / 0, 0 /
c
c     Entry with mfilt = -1 occurs at start of each subintegration loop.
c
       if(nfilts.eq.0) return
       if (mfilt .eq. -1) then
         kfilts = kfiltssv
         if(kfilts07.eq.7) then
           kfilts07=0
         end if
c
c     Entry with mfilt = 0 occurs at start of sinteg routine.
c
c           Increment kfilts.
c
       else if (mfilt .eq. 0) then
       kfilts = mod( kfilts+1, nfilts )
c
c     Entry with mfilt = 1 occurs when sinteg wants to filter a species.
c
c           Do it if kfilts=0.
c
       else if (mfilt .eq. 1) then
         if (kfilts .eq. 0) goto 2000
c
c     Entry with mfilt = -2 occurs at end of stage 3 (within core); at this
c     time set kfiltssv to whatever was attained.
c
       else if (mfilt .eq. -2) then
       kfiltssv = kfilts
       end if
       return
2000   continue
```

Table 6.36, Continued

```
      do 2010 k=1,ktot
        swght(k)=cairclm(k)*swidth(k)
 2010 continue
      filfac=filfacs(1)
      if(filfac.le.0) return
c
c     First filter boundary values if inflow.
c
      lpss = lprops(l)
      if (mbndfils .gt. 0) go to 2100
        pecle1 = (windsclm(1) + vertcclm(1,lpss))
     +  *deltas(1)/amax1( 1.e-10, difusclm(1,1) )
        peclen = (windsclm(ktot) + vertcclm(ktot,lpss))
     +  *deltas(ktot-1)/amax1( 1.e-10, difusclm(ktot,1) )
        call bndfilt( v, ktot, pecle1, peclen,swght,filfac)
c
c    Then do the rest.
c
 2100 continue
      courno = 0.0
      filfac = filfacs(l)
      call foresfilt( v, ktot, filfac,nfilsrch,nfiltset,
     + nfiltitr,swght)
      return
      end
```

Table 6.37: Listing of Subroutine **xfilter**

```
      subroutine xfilter( mfilt, j, k, l )
c
c     Class II subroutine called from subroutine xinteg
c
c     Interfaces to foresfilt and bndfilt for x-dimension.
c
      include 'pluvius.com'
```

Table 6.37, Continued

```
      dimension xwght(imaxd)
      data jfiltx / 0 /
      data ifiltx07 / 7 /
c
c     Call with mfilt = 0 is for adjusting counter.
c
      if(nfiltx.eq.0) return
      if (mfilt .eq. 0) then
        ifiltx = mod( ifiltx+1, nfiltx )
        if(ifiltx07.eq.7) then
          ifiltx07=0
        end if
        return
      else
        if (ifiltx .eq. 0) goto 2000
      end if
 2000 continue
      do 2010 i=1,itot
        xwght(i)=cair(i,j,k)*xwidth(i)
 2010 continue
c
c     First filter boundary values if inflow.
c
      if (mbndfilx .le. 0) goto 2100
      pecle1 = windx(1,j,k)*deltax(1) /
     + amax1( 1.e-10, diffux(1,j,k) )
      peclen = windx(itot,j,k)*deltax(itot-1) /
     + amax1( 1.e-10, diffux(itot,j,k) )
      call bndfilt( v, itot, pecle1, peclen,xwght,filfac)
c
c     Then do the rest.
c
 2100 continue
      courno = 0.0
      filfac = filfacx(l)
      if (filfac .le. 0.) return
      call foresfilt( v, itot, filfac,nfilsrh,
     + nfiltset,nfiltitr,xwght)
      return
      end
```

Table 6.38: Listing of Subroutine **yfilter**

```
      subroutine yfilter( mfilt, i, k, l )
c
c     Class II subroutine called from subroutine yinteg
c
c     Interfaces to foresfilt and bndfilt for y-dimension.
c
      include 'pluvius.com'
      dimension ywght(jmaxd)
      data jfilty / 0 /
      data jfilty07 / 7 /
c
c     Call with mfilt = 0 is for adjusting counter.
c
      if(nfilty.eq.0) return
      if (mfilt .eq. 0) then
        jfilty = mod( jfilty+1, nfilty )
        if(jfilty07.eq.7) jfilty07=0
        return
      else
        if (jfilty .eq. 0) goto 2000
        return
      end if
 2000 continue
      do 2010 j=1,jtot
        ywght(j)=cair(i,j,k)*ywidth(j)
 2010 continue
      filfac=filfacy(1)
      if (filfac.le.0) return
c
c    First filter boundary values if inflow:  first three elements adjacent
c    to each boundary.
c
      if (mbndfily .le. 0) goto 2100
        pecle1 = windy(i,1,k)*deltay(1) /
     +  amax1( 1.e-10, diffuy(i,1,k) )
        peclen = windy(i,jtot,k)*deltay(jtot-1) /
     +  amax1( 1.e-10, diffuy(i,jtot,k) )
        call bndfilt( v, jtot, pecle1, peclen,ywght,filfac)
c
c    Then do the internal elements.
c
 2100 continue
```

Table 6.38, Continued

```
       courno = 0.0
       call foresfilt( v, jtot, filfac,nfilsrch,nfiltset,nfiltitr,ywght)
       return
       end
```

Table 6.39: Listing of Subroutine **bndfilt**

```
       subroutine bndfilt( v, n, pecle1, peclen,wght,filfac)
c
c    Class II subroutine called from subroutines xfilter, yfilter, and sfilter
c
c     Applies filtering at boundaries (first and/or last three grid points) if
c     inflow is detected.
c
c     Subroutine parameters:
c        v - array of species' mixing ratios to be filtered
c        n - dimension of the array
c        pecle1 - peclet number at i=1 boundary
c        peclen - peclet number at i=itot boundary
c        wght - array of grid spacings
c
       dimension v(n), wght(n)
c
c      i=1 boundary:
c      Apply filter if inflow (pecle1>0) and abs(pecle1)>.5
c
       if (pecle1 .le. 0.5) go to 1000
       delv1 = 2.*filfac*(v(2) - v(1))*0.5*(wght(1)+wght(2))
       delv2 = filfac*(v(3) - v(2))*0.5*(wght(2)+wght(3))
       v(1) = v(1) + delv1/wght(1)
       v(2) = v(2) + (delv2 - delv1)/wght(2)
       v(3) = v(3) - delv2/wght(3)
```

Table 6.39, Continued

```
c
c      i=n boundary:
c      Apply filter if inflow (peclen<O) and abs(pecle1)>.5
c
 1000  if (peclen .ge. -O.5) goto 2000
       delvn = 2.*filfac*(v(n) - v(n-1))*.5*(wght(n)+wght(n-1))
       delvnm1 = filfac*(v(n-1) - v(n-2))*.5*(wght(n-1)+wght(n-2))
       v(n) = v(n) - delvn/wght(n)
       v(n-1) = v(n-1) + (delvn - delvnm1)/wght(n-1)
       v(n-2) = v(n-2) + delvnm1/wght(n-2)
 2000  return
       end
```

Table 6.40: Listing of Subroutine **foresfilt**

```
      subroutine foresfilt(c,n,filfac,nfsrch,nfset,maxitr,wght)
c
c    Class II subroutine called from subroutines xfilter, yfilter, and sfilter
c
c     Applies non-linear filter of C.K.Forester (j.comp.phys, 1977, pp 1-22).
c
c----------------------------------------------------------------------------
c
c     Subroutine parameters:
c        c - one dimensional array of values to be filtered
c        n - the number of values
c        filfac - the filter factor =O.5*mu in Forester's paper
c          specified by user for each species.  suggested values are
c               O.1 for filtered species, O.O for non-filtered species
c        wght - one-dimensional array of grid weights
c               wght(i)= air density * grid width
         nfsrch, nfset - n and m of Forester's paper.  A value that is a
c               local maximum (or minimum) is considered to be a true
c               extremum if it exceeds the nfsrch adjacent values.  Otherwise
c               it is treated as "noise," and filtering is applied to it and
c               the nset adjacent values.
```

Table 6.40, Continued

```
c      maxitr - maximum number of filter iterations. The filter is
c             applied maxitr iterations,or until "noise" disappears,
c             whichever occurs first.
c
c-------------------------------------------------------------------------
c     Variables:
c        phifil - phi in forester's paper
c        sfil - s in forester's paper
c
c--------------------------------------------------------------------------- -
c     This code adapted from subroutine filter of pluvius mod 5.0 code
c     by R.C. Easter, 22 Jan. 85
c--------------------------------------------------------------------------- -
       dimension c(*), wght(*)
       parameter (nzmax = 201)
       parameter (nhi = 211)
       parameter (nlo=-9)
       dimension phifil(nlo:nhi), sfil(nlo:nhi), delc(nlo:nhi)
       equivalence (sfil(1),delc(1))
       logical noise
       if (filfac .le. 0.) return
       if(n.gt.(nzmax)) then
         write(6,*) 'array limit exceeded in foresfilt'
         pause
         stop
       end if
       nm1 = n-1
c
c     Iterate maxitr iterations or until noise is gone.
c
       do 4000 nitr = 1, maxitr
c
c     Set phi's to zero and calc. the s's (set values beyond 1 and n to
c     simplify boundary cases).
c
       noise = .false.
       do 1100 i = 2, n
         phifil(i) = 0.
         sfil(i) = sign( 1.0, (c(i)-c(i-1)) )
 1100  continue
       do 1200 i = nlo, 1
         phifil(i) = 0.
         sfil(i) = sfil(2)
```

Table 6.40, Continued

```
1200   continue
       do 1300 i = n+1, n+nfsrch
         phifil(i) = 0.
         sfil(i) = sfil(n)
 1300  continue
c
c     Now check for noisy data points and set phi's as needed.
c
       do 2500 i = 2, nm1
         if (sfil(i) .ne. sfil(i+1)) then
           do 2200 j = 1, nfsrch
             if (sfil(i-j) .ne. sfil(i)) goto 2300
             if (sfil(i+1+j) .ne. sfil(i+1)) goto 2300
 2200      continue
         end if
         goto 2500
2300     do 2400 j = i-nfset, i+nfset
           phifil(j) = 1.0
 2400    continue
         noise = .true.
2500   continue
c
c     Apply filtering if any noisy points found.
c
       if ( noise ) then
         filfachf=0.5*filfac
         do 3300 i=1,nm1
           delc(i)  = filfachf*(c(i+1)-c(i))*(phifil(i+1)+phifil(i))*
     +     (wght(i)+wght(i+1))
 3300    continue
         c(1) = c(1) + delc(1)/wght(1)
         c(n) = c(n) - delc(nm1)/wght(n)
         do 3400 i = 2, nm1
           c(i) = c(i) + (delc(i) - delc(i-1))/wght(i)
 3400    continue
       else
         goto 8000
       end if
4000   continue
8000   return
       end
```

Table 6.41: Listing of Subroutine **chmint**

```
      subroutine chmint(i,j,tlast,dtchem)
c
c     Class II subroutine called from subroutine core
c
      include 'pluvius.com'
c
c      Chmint interfaces between the pluvius-2 core subroutine and the
c      odeint o.d.e. integrator.
c
c      It integrates the o.d.e.'s for local rates of change a single
c      column (i, j fixed but all k) from time tlast to time tlast+dtchem.
c
      do 10 k = 1, ktot
c
c      Convert mixing ratios to concentrations
c        Note: physical variables in call to tfrmth are:
c        pressure, theta-c, cloud+water vapor, fog flag, and temperature.
c        cnn(ltheta) is used to store ****temperatures**** for this step.
c        This temporary transposition of storage locations is required because
c        theta-c is a computed variable and temperature is constrained.
c
        dmycair = cairclm(k)
        do 20 l = 1, ltot
          a(l)=rclm(k,l)*dmycair
          cnn(l)=amax1(rclm(k,l)*dmycair, cmin(l))
 20     continue
        do 30 l = ltot+1, ltot2
          cnn(l)=rclm(k,l)*dmycair
 30     continue
        cnn(ltemp)=rclm(k,ltemp)
        if(mtemp.eq.1) then
c
c      Convert theta-c's to temperatures (for energy bal. integration only).
c
          call tfrmth(ptot(i,j,k),rclm(k,ltheta),rclm(k,lvapcl),
     +    jfog(i,j,k),cnn(ltheta))
          cnn(ltemp)=cnn(ltheta)
        end if
c
c        Make call to odeint.
```

Table 6.41, Continued

```
         dtsug = dtsave(i,j,k)
         call odeint(i,j, k, tlast, dtchem, dtsug, dtchem )
         dtsave(i,j,k) = dtsug
c
c      convert back to mixing ratios
c      if initial conc. was below cmin, adjust downwards
c
         do 40 l = 1, ltot
           if(a(l).lt.cmin(l)) cnn(l) = cnn(l) +a(l) -cmin(l)
           rclm(k,l) = cnn(l)/dmycair
 40      continue
         do 50 l = ltot+1, ltot2
           rclm(k,l) = cnn(l)/dmycair
 50      continue
       rclm(k,ltemp)=cnn(ltemp)
         if(mtemp.eq.1) then
c
c      Convert temperatures back to theta-c's.
c
           rclm(k,ltheta)=(1.e6/ptot(i,j,k))**ak*(cnn(ltheta)-
     +     alocp*rclm(k,lcloud))
         end if
 10    continue
      return
       end
```

Table 6.42: Listing of Subroutine **odeint**

```
       subroutine odeint(i,j,k,tstrto, dttot, dtsug, dtmax )
c
c     Class II subroutine called from subroutine chmint
c
       include "pluvius.com"
```

Table 6.42, Continued

```
c----------------------------------------------------------------------
c
c      subroutine  *****  odeint  *****
c
c      Performs numerical integrations of the system of equations
c
c      d( c(k,l) )/dt  =  ggen(l) - gdec(l)
c
c      where:
c          t is the independent variable
c          c(k,l) is the vector of dependent variables (k is fixed
c             during each call to odeint)
c          ggen(l) is the vector of species generation rates
c          gdec(l) is the vector of species destruction rates.
c             (gdec(l) is expressed internally as a psuedo
c             first-order relation gdec(l) = xi(l)*c(l), where xi(l)
c             is the vector of psuedo first-order rate coefficients.
c
c     Both ggen(l) and gdec(l) are determined by interrogating a user-supplied
c     subroutine, named gen.
c
c
c      tstrto is the model time at the begining of the call to odeint.
c
c      dttot is the time interval over which the equations are integrated.
c
c      dtmax is the (user specified) maximum allowable time step.
c
c      dtsug is the "suggested" initial time step for the integration. If this
c      is set negative by the calling program, then odeint will determine its
c      own step size. (after a successful integration, the final step size is
c      passed back in dtsave(i,j,k).  It should be used to establish
c      dtsug on repeated calls.)
c
c-------------------------------------------------------------------
c
c      The following parameters affect the mechanics of the numerical
c      integration.  they are held in common blocks, and should be initialized
c      in the setup subroutine.
c
c        ncorr is the desired number of corrector passes per time step.  (**an
c        odd number is recommended**).
```

Table 6.42, Continued

```
c
c        iexp determines stiff integration method:  "asymptotic" (iexp = O),
c        or "exponentially assisted" (iexp = 1).
c
c        cmin  =  values of dependent variables, below which they can be
c        considered as insignificant to the overall computation.  Must be
c        set as positive, non-zero numbers prior to subroutine call.
c
c        cerror  =  values of dependent variables, below which the convergence
c        error criterion is relaxed.  Must be set as positive, non-zero
c        numbers prior to subroutine call.  Typically, one should set
c        cmin(k) < .OO1*cerror(k).
c
c        eps  =  maximum allowable normalized disagreement between the
c        predictor and corrector, or between successive corrector cycles;
c        i. e., maximum convergence error.
c
c        ratesf  =  values of psuedo first-order decay coefficient above which
c        the corresponding equations are considered stiff.  This parameter
c        can be adjusted to optimize computationalefficiency of the
c        algorithm.  For typical air chemistry problems, a suggested value
c        is .O1 sec(-1).
c
c        dtmin  =  minimum allowable step size
c
c----------------------------------------------------------------------

      dimension stiff(lmaxd), ffp(lmaxd), cstart(l2maxd),
     + ggenp(lmaxd), xi(lmaxd), xip(lmaxd), pred(lmaxd)
c
c      Initialize variables and flags.
c
c      iflg  =  terminal increment flag;  =  1 when on last
c      timestep of integration.
c      tn  =  current value of dependent variable.
c      dtrem  =  remaining span to be integrated.
c
      iflg = O
      tn = tstrto
      ts = tstrto
      dtrem = dttot
      pfact = 1.
```

Table 6.42, Continued

```
c
c     Determine initial step size: use "suggested value" dtsug if this value
c     is supplied as a positive number in the subroutine call; otherwise
c     compute step size internally.
c
       if (dtsug .gt. 0.) then
         dtau = amin1(dtsug, dttot)
       else
         dtau = dttot
         dtaueff = dtau
         call gen(i,j,k, ts, dtaueff, 1 )
         scrtch = 0.
         scrtch = 1.e-20
         do 10 l = 1, ltot
           if (cnn(l) .ge. 1.1*cmin(l)) then
             dmy1 = .1*eps*abs(ggen(l))-gdec(l)
             dmy2 = sign(1./cnn(l), dmy1)
             dmy1 = dmy2*gdec(l)
             dmy2 = -abs(abs(ggen(l))-gdec(l))*dmy2
             dmy1 = amax1(dmy1, dmy2)
             scrtch = amax1(dmy1, scrtch, 1.e-20)
           end if
 10      continue
         dtau = .5/scrtch
         dtau = amin1(dtau, dtmax)
       endif
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c
c      Begin an integration step for step-size dtau.
c
c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 20    dtrem = dtrem-dtau
       pfact = pfact+.1*(1.-pfact)
       nneg = 0
c
c     (nneg is flag which is set to 1 when a negative value of the dependent
c     variable is detected.  This is followed by a srinkage of step size and a
c     restart).
c
c      Check for runaway computation (dtau .lt. dtmin).
```

Table 6.42, Continued

```
       if (dtau .lt. dtmin) then
         write(6,2000) k, ts ,dtau,dtmin
         pause
         stop 99
       end if
c
c      Check for final subincrement in integration. Preserve timestep and set
c     flag if final.
c
       if (dtrem .le. 0.) then
         iflg = 1
         dtsug = dtau
         dtau = dtau+dtrem
         dtrem = 0.
       endif
c
c      Save current values of dependent variables in case step has to be
c      repeated.
c
       do 30 l = 1, ltot2
         cstart(l) = cnn(l)
 30    continue
c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c      predictor phase
c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c      Set generation and decay rates.
c
       jpasso = 1
       dtaueff = dtau
       call gen(i,j,k,ts,dtaueff,jpasso)
c
c      Set current time.
c      Check for stiffness and set flags.
c
       do 40 l= 1, ltot
         ggenp(l) = ggen(l)
         cstart(l) = cnn(l)
         xip(l) = gdec(l)/amax1(cnn(l),cmin(l))
```

Table 6.42, Continued

```
         stiff(l) = xip(l)-ratesf(l)
         stiff(l) = .5+sign(.5, stiff(l))
         ffp(l) = ggen(l)-gdec(l)
         if (iexp .eq. 0) then
           cnn(l) = dtau*ffp(l)/(1.+xip(l)*dtau*stiff(l))+cnn(l)
         else
           if (stiff(l) .gt. .99) then
             dmy1 = xip(l)*dtau
             exp1 = 1.
             if (dmy1 .lt. 1.e30) exp1 = 1.-exp(-dmy1)
             cnn(l) = ffp(l)*exp1/xip(l)+cnn(l)
           else
             cnn(l) = ffp(l)*dtau+cnn(l)
           endif
         endif
c
c      Check for bad convergence -- bypass calculations if bad.
c
         if (cnn(l) .lt. 0..and.stiff(l) .lt. .99) then
           nneg = 1
           ierrqq=l
           pfact=.2
           go to 90
         end if
c
c      Set historical predictor values.
c
         cnn(l) = amax1(cnn(l), cmin(l))
         pred(l) = cnn(l)
 40    continue
c      if(j.eq.15.and.k.eq.7) then
c      write(6,*) 'predictor: cnn,ggen,gdec,dtau'
c      write(6,*) cnn(2),ggen(2),gdec(2),dtau
c      end if
       if(nneg.eq.0) then
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c            corrector phase
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccc
         ts = tn+dtau
```

Table 6.42, Continued

```
         do 60 icorr=1,ncorr
           jpasso = jpasso + 1
           call gen(i,j,k,ts, dtaueff, jpasso )
           do 70 l = 1, ltot
             xi(l) = gdec(l)/amax1(cnn(l),cmin(l))
             xim = .5*(xip(l)+xi(l))
             ggenm = .5*(ggen(l)+ggenp(l))
             if (stiff(l) .lt. .99) then
c
c      Not stiff; use conventional Euler's method.
c
               cnn(l) = cstart(l)+dtau*(ggenm-.5*xim*(cnn(l)+cstart(l)))
c
c      Check for bad convergence; bypass calculations if bad.
c
               if (cnn(l) .lt. 0..and.stiff(l) .lt. .99) then
                 nneg = 1
                 ierrqq=l
                 pfact=.2
                 go to 90
               end if
             else
c
c      Stiff; use exponential approximation.
c
               if (iexp .eq. 0) then
c
c      Use asymptotic form.
c
                 dmy1 = dtau*(1.+.5*xim*dtau)
                 cnn(l) = cstart(l)+(ggenm-cstart(l)*xim)*
     +           dmy1/(1.+xim*dmy1)
               else
c
c      Use exponential form
c
                 dmy1 = xim*dtau
                 exp1 = 1.
                 if (dmy1 .lt. 1.e30) exp1 = 1.-exp(-dmy1)
                 cnn(l) = (ggenm/xim-cstart(l))*exp1+cstart(l)
               end if
             end if
```

Table 6.42, Continued

```
             cnn(l) = amax1(cnn(l), cmin(l))
 70        continue
 60      continue
c      if(j.eq.15.and.k.eq.7) then
c      write(6,*) 'corrector: cnn,ggen,gdec,dtau ='
c      write(6,*) cnn(2),ggen(2),gdec(2),dtau
c      pause
c      end if
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c      error check and step size adjustment
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c      Basic convergence criterion is:
c          abs( (predictor-corrector)/corrector ) < eps
c
c      The convergence criterion is relaxed under a number of circumstances.
c      This error relaxation should be considered "application-dependent", and
c      the user may wish to modify it as required.  in the air-chemistry
c      applications of the Pluvius Mod 5 users' manual, errors are relaxed
c      whenever:
c          1) concentrations approach negligible values (rlocon)
c          2) the starting concentration is negligibly small (rzrost)
c          3) the projected increase is so large as to make the current
c             value and its relative error small by comparison (rfasti)
c          4) the projected decrease is rapid (rfastd)
c
c      Set minimum value of errmx
c
           errmx = 1.e-6
c
c      Setting minimum errmx to 1.e-6 limits step size increase to
c      sqrt(1.e6*eps).
c
         do 80 l = 1, ltot
           rlocon = 0.5*(cstart(l) + cnn(l))/cerror(l)
           rzrost = abs( cstart(l)/cmin(l) - 1.)
           projtn = cstart(l) + dtrem*ffp(l)/(1.+xip(l)*dtrem*stiff(l))
           if (projtn .gt. 0.) then
             rfasti = cnn(l)/projtn
```

Table 6.42, Continued

```
            rfastd = projtn/cmin(l)
          else
            rfasti = 1.
            rfastd = 0.
          end if
          redftr = amin1(1., rlocon, rfastd, rfasti, rzrost)
          err = (cnn(l)-pred(l))/amax1(cnn(l),pred(l))
          erelax = abs(err)*redftr
          if (erelax .gt. errmx) ierrqq = l
          errmx = amax1(erelax, errmx)
80        continue
      end if
      if(errmx.lt.eps) go to 105 !jump to 105 in event of acceptable error
c
c     If errmx < eps, then error exceeds tolerance.
c     Apply penalty factor and set up data for restart.
c     If negative values were experienced (nneg=1), halve the stepsize.
c     Otherwise, compute new step size based on size of error.
c
        pfact=.4
 90     continue
c        write(6,666) i,j,k,ierrqq,nneg,dtau,cnn(ierrqq),dtrem
c     write(6,*) 'ggen, gdec = ',ggen(ierrqq), gdec(ierrqq)
c666    format(1h ,'failure at i,j,k,l, nneg = ',5i5,/,
c    +   'dtau, cnn,dtrem = ',3e10.2)
        ts = tn
        do 100 l = 1, ltot2
          cnn(l) = cstart(l)
 100    continue
        dtrem = dtrem+dtau
        iflg = 0
        if (nneg .eq. 1) then
          dtau = pfact*dtau*0.5
        else
          dtau = dtau * 0.8 * pfact * sqrt(eps/errmx)
          dtau = amin1(dtau, dtmax)
        end if
c
c     Error too large: recycle with smaller step size to reduce it.
c
      go to 20
c
```

Table 6.42, Continued

```
c      Otherwise, error was acceptable; proceed to integrate next step.
c
 105   continue
c
c       But first check for successful completion of current span (iflg=1);
c      return to calling routine if complete.
c
         if (iflg .eq. 1) go to 110
c
c       If not complete, compute new step size and proceed.
c
         dtau = dtau * 0.8 * pfact * sqrt(eps/errmx)
         dtau = amin1( dtau, dtmax )
         tn = ts
        go to 20
 2000  format(/' *** step size below minimum value in odeint1 ***'/
     +     ' grid point =', i3/' model time =', e12.4,
     +      'dtau,dtmin =',2e10.3)
 110   return
       end
```

Table 6.43a: Listing of Subroutine **gen** for Case Example A

```
      subroutine gen(i,j,k,igcalc,tgen,dtau,jpasso)
c
c     Class I subroutine called from subroutine odeint
c
c     Called repeatedly in both the predictor and corrector phases of
c     subroutine ***odeint***.
c
c     Subroutine ***gen*** computes generation (ggen(l)) and decay (gdec(l))
c     rates in moles/cm3 sec for advected species 1 through ltot.
c     Note:  gdec values are positive, and total rate of change is ggen-gdec.
c
c      tgen is current model time.
c      dtau is current time step in odeint.
```

Table 6.43a, Continued

```
c      jpasso is flag specifying predictor (1) or corrector (2,3,...) pass at
c      current interrogation point in odeint.
c

c
      include 'pluvius.com'
      data xnc /500./
c
      do l=1,ltot
        ggen(l)=0.
        gdec(l)=0.
      end do
c
c     define local variables
c
      temp=cnn(ltheta)
      tempc=temp-273.2
      vsatw=sat(temp)
      vapor=amin1(cnn(lvapcl),vsatw)
      cloud=cnn(lvapcl)-vapor
      jfog(i,j,k)=1
      if(cloud.le.0.) then
      jfog(i,j,k)=0
      cloud=0.
      end if
      if (tempc.le.0.) then
        vsati=ssat(temp)
        else
        vsati=vsatw
      end if
      cnn(lwvap)=vapor
      cnn(lcloud)=cloud
      rain=cnn(lrain)
      snow=cnn(lsnow)
      cnn(lwvap)=vapor
c
c     Cloud physics block:  convert to g/m3 units and initialize rates to 0.
c
      xcloud=cloud*1.8e7
      xrain=rain*1.8e7
      xsnow=snow*1.8e7
      xvapor=vapor*1.8e7
```

Table 6.43a, Continued

```
      auto=0.
      accr=0.
      evap=0.
      freez=0.
      smel=0.
      rim=0.
      deposi=0.
      sublim=0.
c
c     Autoconversion rate = auto*cloud.
c     Accretion rate = accr*cloud.
c
      if(cloud.gt.0.) then
        xcdrop = 1.e-6*xcloud/xnc
        if(xcdrop.gt.2.348e-9) auto=5.504e9*xcloud*
     +   (xcdrop**1.667)*(1.-(2.347e-9/xcdrop)**1.333)*
     +   (1.-(2.040e-9/xcdrop)**.333)
        accr=3.39e-3*(xrain**.875)
      else
c
c     Evaporation rate = evap*rain.
c
        evap=5.44e-4*(vsatw*1.8e7-xvapor)/
     +   (amax1(xrain,1.e-25)**.35)
        evap=amax1(0.,evap)
        evap=amin1(1.,evap)
      end if
c
c     Riming growth rate = rim*cloud.
c     y0flak = n0,s from marshall-palmer distribution.
c     slambd = lambda from marshall-palmer distribution.
c
      y0flak = 5.681e6*exp(-.0878*amin1(tempc,0.))
      slambd = 5.63*(y0flak/(xsnow+1.e-20))**0.3333
      if((xsnow.ge.1.e-12).and.(cloud.gt.0.)) then
        a0lamb = 3.50e-4*slambd
        rim = 0.319*y0flak*slambd**(-3.206)*
     +   exp(-a0lamb)*(a0lamb*(a0lamb+2.)+2.)
      end if
c
c     For tempc < 0 calculate:
c     depositional growth rate = deposi
```

Table 6.43a, Continued

```
c     sublimation rate = subli
c     freezing rate = freez*rain
c
c     In calc's for small modisperse ice-crystal case:
c     ycryst = crystals per cubic meter of space
c     xcryst = individual crystal mass (gm)
c     depcry = depositional growth rate for the crystals
c
      if(tempc.lt.O.) then
        ycryst=amax1(.OO6,4OO.*((-tempc/16.)**4.))
        xcryst=amax1(xsnow/ycryst,6.Oe-11)
        difh2o=.226e-4*((temp/273.15)**1.81)*
     +   (1.e6/ptot(i,j,k))
        gsi=.82*difh2o*(xvapor-vsati*1.8e7)
        depcry=ycryst*(.196*xcryst**.38)*gsi*
     +   (1.+374.*aflake**.75)
        if(xsnow.ge.1.e-12) then
          aOlamb=amax1(O.,alog(yOflak/(slambd*ycryst)))
          ytflak=(yOflak/slambd)*exp(-aOlamb)
          aflake=(1.+aOlamb)/slambd
          deposi=ytflak*(6.28*aflake)*gsi*
     +     (1.+374.*aflake**.75)
        end if
        deposi=amax1(deposi,depcry)/1.8e7
        if(deposi.lt.O.) then
          sublim=-deposi/snow
          sublim=amin1(1.,sublim)
          deposi=O.
        end if
        freez = 2.57e-5*ycryst*(xrain*.63)
        freez = amin1(1.,freez)
      else
c
c     For tempc > O calculate:
c     melting rate = smel*snow
c     Also, riming rate transfers cloud water to rain category rather than to
c     snow in this case.
c
        rmelt=(tempc-4.677e7*(2.693e-7-vapor))*
     +   (7.492e-6*(snow*.667) + 7.415e-4*(snow**.868))
     +   +1.226e-2*tempc*rim*cloud
        smel=rmelt/amax1(snow,1.e-25)
```

Table 6.43a, Continued

```
          if(smel.lt.O.) smel=O.
          smel=amin1(1.O,smel)
          accr=accr+rim
          rim=O.
        end if
c
c       Calculate ggen and gdec terms for water species.
c
        ggen(lrain) = (auto+accr)*cloud + smel*snow
        gdec(lrain) = (evap+freez)*rain
        ggen(lsnow) = deposi + rim*cloud +freez*rain
        gdec(lsnow) = (smel+sublim)*snow
        ggen(lvapcl) = evap*rain + sublim*snow
        gdec(lvapcl) = (auto+accr+rim)*cloud +deposi
c
c       Now calculate latent heating terms for theta-c equation
c
        if(jfog(i,j,k).eq.O) then
          ggen(ltheta) = (-heatc*evap*rain
     +    + (heatc+heatf)*deposi
     +    + heatf*freez*rain)/(cp*cair(i,j,k))
        else
          chi=1.+vapor*heatc*heatc/(cp*glc*temp*temp*cair(i,j,k))
          ggen(ltheta) = heatf*(deposi+freez*rain)/(chi*cp*cair(i,j,k))
        end if
        return
        end
```

Table 6.43b: Listing of Subroutine **gen** for Case Example B

```
         subroutine gen(i,j,k,tgen,dtau,jpasso)
c
c        Class I subroutine called from subroutine odeint
c
c         Called repeatedly in both the predictor and corrector phases of
c         subroutine ***odeint***.
```

Table 6.43b, Continued

```
c
c      Subroutine ***gen*** computes generation (ggen(l)) and decay gdec((l))
c      rates in moles/cm3 sec for advected species 1 through ltot.
c
c      Note: gdec values are positive, and total rate of change is ggen-gdec.
c
c       tgen is current model time.
c       dtau is current time step in odeint.
c       jpassO is flag specifying predictor (1) or corrector (2,3,...).
c       pass at current interrogation point in odeint.
c
       include 'pluvius.com'
       data xnc /5OO./
       data ijkold /O/
       ijknew = i+j+k
       if(ijknew.eq.ijkold) go to 1OO
       ijkold=ijknew
c
       do l=1,ltot
              ggen(l)=O.
              gdec(l)=O.
       end do
c
c      Define local variables.
c
       temp=cnn(ltemp)
       tempc=temp-273.2
       vsatw=sat(temp)
       vapor=amin1(cnn(lvapcl),vsatw)
       if (tempc.le.O.) then
              vsati=ssat(temp)
       else
              vsati=vsatw
       end if
       rain=cnn(lrain)
       snow=cnn(lsnow)
       cloud=cnn(lcloud)
c
c      Cloud physics block: convert to g/m3 and initialize rates.
c
       xcloud=cloud*1.8e7
       xrain=rain*1.8e7
```

Table 6.43b, Continued

```
       xsnow=snow*1.8e7
       xvapor=vapor*1.8e7
       auto=O.
       accr=O.
       evap=O.
       freez=O.
       smel=O.
       rim=O.
       deposi=O.
       sublim=O.
c
c      Autoconversion rate = auto*cloud.
c      Accretion rate = accr*cloud.
c
       if(cloud.gt.O.) then
              xcdrop = 1.e-6*xcloud/xnc
              if(xcdrop.gt.2.348e-9) auto=5.5O4e9*xcloud*
     +        (xcdrop**1.667)*(1.-(2.347e-9/xcdrop)**1.333)*
     +        (1.-(2.O4Oe-9/xcdrop)**.333)
              accr=3.39e-3*(xrain**.875)
       else
c
c      Evaporation rate = evap*rain.
c
              evap=5.44e-4*(vsatw*1.8e7-xvapor)/
     +        (amax1(xrain,1.e-25)**.35)
              evap=amax1(O.,evap)
              evap=amin1(1.,evap)
       end if
c
c      Riming growth rate = rim*cloud.
c      yOflak = nO,s from marshall-palmer distribution.
c      slambd = lambda from marshall-palmer distribution.
c
       yOflak = 5.681e6*exp(-.O878*amin1(tempc,O.))
       slambd = 5.63*(yOflak/(xsnow+1.e-2O))**O.3333
       if((xsnow.ge.1.e-12).and.(cloud.gt.O.)) then
         aOlamb = 3.5Oe-4*slambd
         rim = O.319*yOflak*(slambd**(-3.2O6))*exp(-aOlamb)*
     +   (aOlamb*(aOlamb+2.)+2.)
       end if
```

Table 6.43b, Continued

```
c      For tempc < O calculate:
c      depositional growth rate = deposi
c      sublimation rate = subli
c      freezing rate = freez*rain
c
c      In calc's for small modisperse ice-crystal case:
c      ycryst = crystals per cubic meter of space
c      xcryst = individual crystal mass (gm)
c      depcry = depositional growth rate for the crystals
c
       if(tempc.lt.O.) then
              ycryst=amax1(.OO6,4OO.*((-tempc/16.)**4.))
              xcryst=amax1(xsnow/ycryst,6.Oe-11)
              difh2o=.226e-4*((temp/273.15)**1.81)*
     +        (1.e6/ptot(i,j,k))
              gsi=.82*difh2o*(xvapor-vsati*1.8e7)
              depcry=ycryst*(.196*xcryst**.38)*gsi*
     +        (1.+374.*aflake**.75)
              if(xsnow.ge.1.e-12) then
              aOlamb=amax1(O.,alog(yOflak/(slambd*ycryst)))
              ytflak=(yOflak/slambd)*exp(-aOlamb)
              aflake=(1.+aOlamb)/slambd
              deposi=ytflak*(6.28*aflake)*gsi*
     +        (1.+374.*aflake**.75)
              end if
              deposi=amax1(deposi,depcry)/1.8e7
              if(deposi.lt.O.) then
              sublim=-deposi/snow
              sublim=amin1(1.,sublim)
              deposi=O.
              end if
              freez = 2.57e-5*ycryst*(xrain*.63)
              freez = amin1(1.,freez)
       else
c
c       For tempc > O calculate:
c       melting rate = smel*snow
c       Also, riming rate transfers cloud water to rain category rather than
c       to snow in this case.
c
              rmelt=(tempc-4.677e7*(2.693e-7-vapor))*
```

Table 6.43b, Continued

```
     +       (7.492e-6*(snow*.667) + 7.415e-4*(snow**.868))
     +       +1.226e-2*tempc*rim*cloud
             smel=rmelt/amax1(snow,1.e-25)
             if(smel.lt.O.) smel=O.
             smel=amin1(1.O,smel)
             accr=accr+rim
             rim=O.
       end if
c
c      Calculate ggen and gdec terms for pollutants.
c
c      initialize variables
c
 1OO   xno3g=cnn(lno3gcl)
       so2g=cnn(lso2gcl)
       ozoxcl=O.
       ozoxrn=O.
       peroxcl=O.
       peroxrn=O.
       so4cl=amax1(O.,(cnn(lso4gcl)-cnn(lso4g)))
       if(cloud.eq.O.) so4cl=O.
       so2cl=O.
       rtnuclso4=O.
       ssscav=O.
       rsscav=O.
       rtnuclno3=O.
       xno3cl=O.
       snscav=O.
       rnscav=O.
       xh2o2cloud=O.
       hno3in=O.
       hno3out=O.
       perin=O.
       perout=O.
       so2in=O.
       so2out=O.
       I=1
c
c      block 1: equilibrium calcs
c
c       Use approximate ion balance to estimate cloud and rain acidity; racid
c       and cacid have units of moles h per liter of water, and are used
```

Table 6.43b, Continued

```
c       internally for solubility calc's.  cnn(lhcloud) and cnn(lhrain) have
c       the normal odeint units of moles per cc total space; these are
c       converted back to the r and rclm arrays (mixing ratios) in the pluvius
c       superstructure.
c
c       Acidity calculations for rain and cloud water.
c       Note:  the ion balance approximation has been taken from the MAP3S
c       Atmospheric Environment article 16, 1603, (1982).
c
       cacid=1.e-5   ! molar; place-keeper for zero-cloud conditions
       if(cloud.gt.0.) cacid=1.03e-5+55.55*((so4cl)
     + *1.184 + cnn(lno3gcl)*.767)/cloud
       cnn(lhcloud)=cacid*cloud*0.018
       racid=1.e-5   ! molar; place-keeper for zero-rain conditions
       if(rain.gt.0.)  racid=1.03e-5+55.55*(cnn(lso4rain*1.184)
     +  + cnn(lno3rain)*0.767)/rain
       cnn(lhrain)=racid*rain*0.018
c
c      solubility computations for cloud and rain water
c
c      cloud so2:
c
c       Calculate solubility coefficient hprime on the basis of local gas-
c       phase concentration and cloud acidity (see subroutine for def. of
c       variables.).
c
       patm=ptot(i,j,k)/1.013e6
       so2gcltot=(cnn(lso2gcl))/cair(i,j,k) !mole fraction
       call henry(hprime,so2gcltot,cacid,temp,
     + patm,0)
c
        gcltot=cnn(lso2gcl)
       so2cl= 18.*gcltot*cloud*hprime/
     + (18.*cloud*hprime+cair(i,j,k))
       so2g=cnn(lso2gcl)-so2cl
c
c      These are is the gaseous and cloud water concentrations of dissolved
c       SO2 in moles/cc space.
c
c       Do solubility calculations for peroxide and undiss. s(iv) in cloud for
c       ox. rate calc's.
c
```

Table 6.43b, Continued

```
      solh=8.2414e-6*exp(6834.93/temp)   !moles/liter-atmosphere
      sols=exp(-10.21+3112./temp) !moles/liter atmosphere
      patmcs=patm/cair(i,j,k)
      solh=solh*patmcs*.001       !dimensionless units
      aqh=1000.*solh*cnn(lh2o2gcl)/(1.+18.*solh*cloud)
      aqs=sols*(so2g)*patmcs
      xh2o2cloud=aqh*.018*cloud   ! distributed cloudwater peroxide conc.
c
c      aqh and aqs are, respectively, the molar concentrations of
c      peroxide and undissociated s(iv) (moles/liter cloud water).
c
c      Compute distributed solubility parameter for rainborne h2o2 (for
c      mass-transfer calc's below).
c
      if(rain.gt.0.) then
      xip=18.*solh*rain
c
c      Compute disperse-phase SO2 solubility parameter for rain (to be used
c      below for mass-transfer calc's).
c
      crainso2=(cnn(lso2rain))/(18.*rain)
      call henry(hprime,crainso2,racid,temp,patm,1)
      rso2eq=crainso2*hprime
      cso2eq=rso2eq*cair(i,j,k)
      xis=cnn(lso2rain)/cso2eq
      end if
c
c     Nitric acid solubility calculations:
c
c     Compute distributed HNO3 solubility in cloud water:
c
      if (cnn(lhcloud).gt.0.) then
      xin=4.96e-6*cloud*cloud*exp(8706./temp)/cnn(lhcloud)
      xno3cl=xin*cnn(lno3gcl)/(1.+xin)
      hno3g=cnn(lno3gcl)-xno3cl
      else
      xno3cl=0.
      hno3g=cnn(lno3gcl)
      end if
c
c      This is the cloud water concentration of nitrate ion in moles/cc
c      space.
c
```

Table 6.43b, Continued

```
c       Compute distributed hno3 solubility in rain water (for mass-transfer
c       calculations, below).
c
       if (cnn(lhrain).gt.0.) then
       xin=4.96e-6*rain*rain*exp(8706./temp)/cnn(lhrain)
       end if
c
c      block 2:  rate calculations
c
c      cloud calculations
c
       if(cnn(lcloud).gt.0.) then
       ozoxcl=2.9e6*(so2cl)*cnn(lozone)/sqrt(cacid)
c      ozone reaction rate in cloud:
       peroxcl=1.5e4*aqh*aqs*cloud
c      peroxide reaction rate in cloud (moles/sec cc space)
       end if
c
c      rain calculations
c
       if(cnn(lrain).gt.0.) then
       ozoxrn=2.96e6*cnn(lso2rain)
     + *cnn(lozone)/sqrt(racid)
c
c      ozone reaction rate in rain (moles/sec cc space)
c
       ek1=exp(-10.96+1970./temp)
       aqs= 55.5*racid*cnn(lso2rain)
     + /((racid+ek1)*cnn(lrain))
       aqh=55.5*cnn(lh2o2rain)/cnn(lrain)
       peroxrn=1.54e4*aqs*aqh*cnn(lrain)
c
c      peroxide reaction rate in rain (moles/sec cc space)
c
c      gaseous mass transfer to rain
c
       trcoeff=1.35*rain/cair(i,j,k)
c
c      general mass transfer coefficient
c
       hno3in=trcoeff*hno3g
       hno3out=trcoeff*cnn(lno3rain)/xin
```

Table 6.43b, Continued

```
       perin=trcoeff*peroxg
       perout=trcoeff*cnn(lh2o2rain)/xip
       so2in=trcoeff*so2g
       so2out=trcoeff*cnn(lso2rain)/xis
c
       end if
c
c      mass transfer rates of gas to rain (moles/cc space sec)
c
c      Compute aerosol nucleation rates.
c
       rnuc=0.
       if(cloud.gt.0..and.winds(i,j,k).gt.0.) rnuc=.01
       rtnuclso4=rnuc
       rtnuclno3=rnuc
c
c      nucleation rates of sulfate and nitrate (moles/cc space sec)
c
c      Compute snow scavenging rates.
c
       ssscav=-90.*snow*vertcclm(k,lprops(lsnow))
       snscav=-90.*snow*vertcclm(k,lprops(lsnow))
c
c      Compute rain scavenging rates
c
       rsscav=-36.*vertcclm(k,lprops(lrain))*rain
       rnscav=-36.*vertcclm(k,lprops(lrain))*rain
c
c      These are scavenging coefficients (sec-1).
c
       ggen(lso2gcl)=so2out
       gdec(lso2gcl)=peroxcl+ozoxcl+
     + (accr+auto+rim)*so2cl+so2in
       ggen(lso2rain)=(auto+accr)*so2cl+so2in+
     + smel*cnn(lso2snow)
       gdec(lso2rain)=freez*amax1(0.,cnn(lso2rain))
     + +peroxrn+ozoxrn+so2out
       ggen(lso2snow)=rim*so2cl+freez*cnn(lso2rain)
       ggen(lso4g)=0.
       gdec(lso4g)=rtnuclso4*cnn(lso4g)+(ssscav+rsscav)
     + *cnn(lso4g)
       ggen(lso4gcl)=peroxcl+ozoxcl
       gdec(lso4gcl)=(rim+accr+auto)*so4cl +
```

Table 6.43b, Continued

```
     + (ssscav+rsscav)*cnn(lso4g)
       ggen(lso4rain)=peroxrn+ozoxrn+rsscav*cnn(lso4g)+
     + smel*cnn(lso4snow)+(auto+accr)*so4cl
       gdec(lso4rain)=freez*cnn(lso4rain)
       ggen(lso4snow)=ssscav*cnn(lso4g)+freez*cnn(lso4rain)+
     + rim*so4cl
       gdec(lso4snow)=smel*cnn(lso4snow)
       ggen(lno3gcl)=rtnuclno3*cnn(lno3g)+hno3out
       gdec(lno3gcl)=(rim+accr+auto)*xno3cl+hno3in
       ggen(lno3g)=O.
       gdec(lno3g)=(snscav+rnscav)*cnn(lno3g)+
     + rtnuclno3*cnn(lno3g)
       ggen(lno3rain)=(auto+accr)*xno3cl+rnscav*cnn(lno3g)+
     + smelt*cnn(lno3snow)+hno3in
       gdec(lno3rain)=freez*cnn(lno3rain)+hno3out
       ggen(lno3snow)=snscav*cnn(lno3g)+freez*cnn(lno3rain)
       gdec(lno3snow)=smel*cnn(lno3snow)
       ggen(lh2o2gcl)=perout
       gdec(lh2o2gcl)=(auto+accr+rim)*xh2o2cloud+peroxcl+
     + perin
       ggen(lh2o2rain)=(auto+accr)*xh2o2cloud+perin+
     + smel*cnn(lh2o2snow)
       gdec(lh2o2rain)=peroxrn+freez*cnn(lh2o2rain)+perout
       ggen(lh2o2snow)=rim*xh2o2cloud+freez*cnn(lh2o2rain)
       gdec(lh2o2snow)=smel*cnn(lh2o2snow)
       ggen(lozone)=O.
       gdec(lozone)=ozoxcl+ozoxrn
       ijkold=ijknew
       return
       end
```

Table 6.44: Listing of Function **aircon**

```
      function aircon( p, tempk )
c
c     Class I function to compute molar air density using gas-law
c     equation.  Called from a variety of subroutines.
c
c      aircon = air density (moles/cc)
c      p      = pressure (dynes/cm**2)
c      tempk  = temperature (K)
c
      aircon = p/(8.3144e7*tempk)
      return
      end
```

Table 6.45: Listing of Function **esat**

```
      function esat( tk )
c
c     Class II function called from cloud-physics routines
c
c     Internal function to compute saturation vapor pressure of water vapor
c     over liquid water, based on polynomial fit to observed data.
c
c      esat = saturation vapor pressure over water (dynes/cm2) at temperature
c      tk (K).
c      For -40<tc< 50, the j-fraction approximation has maximum error of .06%
c      Below -40, a simple form of the Clausius-Clapeyron equation is used.
c
      dimension a(7)
      data a/ -.212929759e2, -.762901437e4, -.177013938e3,
     +          .107023730e5,  .181207195e1, .247985588e4,
     +         -.231047743e2 /
      if (tk .ge. 233.16) then
        tc = tk-273.16
        esat = a(1) + a(2)/(a(3) + tc + a(4)/
     +  (a(5) + tc + a(6)/(a(7) + tc)))
      else
c       esat = 0.18903 * exp( 5709.18/233.16 - 5709.18/tk )
        esat = exp( 22.820254 - 5709.18/tk )
      end if
      esat=esat*1.0e3
      return
      end
```

Table 6.46: Listing of Function **esati**

```
      function esati( tk )
c
c     Class II function called from cloud-physics routines
c
c     Computes saturation vapor pressure of water vapor over ice, based on fit
c     to observed data.
c
c      esati = saturation vapor pressure over ice (dynes/cm2) at temperature
c      tk (K).
c      For -50< tc<50, the j-fraction approximation has maximum error of .02%
c      Below -50, a simple form of the Clausius-Clapeyron equation is used.
c
      dimension a(7)
      data a/ -.136076540e1, -.340752185e3, -.970586093e2,
     +           .532049788e4,  .268793249e2, .113134694e4,
     +           .147762227e2 /
      if (tk .ge. 223.16) then
        tc = tk-273.16
        esati = a(1) + a(2)/(a(3) + tc + a(4)/
     +   (a(5) + tc + a(6)/(a(7) + tc)))
      else
c       esati = 0.039343 * exp( 6150.09/223.16 - 6150.09/tk )
        esati = exp( 24.32367 - 6150.09/tk )
      end if
      esati=esati*1.0e3
      return
      end
```

Table 6.47: Listing of Function **sat**

```
      function sat (tk)
c
c     Class II function called from a variety of subroutines:
c     Computes concentration of saturated water vapor over
c     liquid water, using gas-law equation.
c      sat = saturation vapor concentration (moles h2o/cm**3) over water at
c     temperature tk (K).
c
      sat = esat(tk) / (8.3144e7 * tk)
      return
      end
```

Table 6.48: Listing of Function **ssat**

```
       function ssat (tk)
c
c     Class II function called from a variety of subroutines.
c     Function to compute equilibrium concentration of water vapor
c     over ice, using gas-law equation.
c      ssat = saturation vapor concentration (moles h2o/cm**3)
c      over ice at temperature tk (K).
c
       ssat = esati(tk)/ (8.3144e7 * tk)
       return
       end
```

Table 6.49: Listing of Function **rlapse**

```
       function rlapse ( tempk, cair,jfog )
c
c     Class II function called from subroutine init.
c     Calculates the adiabatic lapse rate under wet or dry
c     conditions, based on hydrostatic equation.
c
c      rlapse = lapse rate (deg. K/cm)
c      adiabatic if jfog=O
c      pseudoadiabatic otherwise
c      tempk = temperature (K)
c      cair  = air density (moles/cc)
c
c      Thermodynamic units are cgs (constants use moles).
c      The sat function must return the saturation vapor density (moles/cc) at
c     temperature tempk.
c
       data gammad/-9.76e-5/! dry adiabatic lapse rate (deg k/cm)
       data cp/2.91e8/ ! air heat capacity, constant pressure (erg/mole/deg k)
       data r/8.314e7/      ! gas constatn (erg/mole/deg k)
       data hlc/4.5e11/     ! latent heat of condensation (erg/mole water)
       rlapse = gammad
       if (jfog.ne.O) then
         dummy1 = hlc*sat(tempk)/(cair*r*tempk)
         rlapse = gammad*(1. + dummy1)/(1. + dummy1*hlc/(cp*tempk))
       end if
       return
       end
```

Table 6.50: Listing of Subroutine **henry**

```
       subroutine henry ( hprime, conc, hex, t, p, lgopt )
c
c     Class III subroutine called from subroutine gen
c
c     Subroutine for calculating apparent henry's-law constant on the basis of
c     Johnstone-Leppla parameters, using the equations of Hales and Sutter.
c
c     hprime is the apparent henry's-law constant, defined by:
c           hprime = so2_gas / so2_aq,  where:
c          so2_aq = concentration of sulfur-IV in aqueous phase
c                    (moles/cm**3 water)
c          so2_gas = concentration of gas phase so2 (mole fraction)
c     c is either
c        so2_aq  when lgopt = 1  (moles/cm**3 water)
c        so2_gas when lgopt = 0  (mole fraction)
c     hex is the aqueous phase hydrogen ion concentration from species
c           other than sulfur-IV (molar)
c     t is temperature (degrees k)
c     p is atmospheric pressure (atmospheres)
c     lgopt is a flag explained above
c
       if ((lgopt.ne.0) .and. (lgopt.ne.1)) then
        write(*,*) 'Bad lgopt in subroutine henry'
        stop
       end if
       hank = exp(10.21-3112./t)/p
       eqcon = exp(-10.96+1970./t)
       b=eqcon+hex
       if(lgopt.ne.1) go to 20
c
c      lgopt=1 --> conc is the aqueous concentration
c
       cdum=eqcon*1000.*conc
       test1=4.*cdum/(b*b)
       if(test1.gt..001) go to 10
       hprime=hank*(1.-eqcon/b)
       go to 40
 10    hprime = hank*(1.+(b-sqrt(b*b+4.*cdum))/(2000.*conc))
       go to 40
c
c      lgopt=0 --> conc is the gas phase concentration
c
```

Table 6.50, Continued

```
 20    test1 = 4.*eqcon*conc/(hex*hex*hank)
       if(test1 .gt. .001) go to 30
       hprime=hank*hex/b
       go to 40
 30    hprime=conc/(conc/hank-
     + (hex-sqrt(hex*hex+4.*conc*eqcon/hank))/2.)
 40    hprime = hprime*1000.
       return
       end
```

Table 6.51: Listing of Subroutine **tfrmth**

```
       subroutine tfrmth(p,th,rvc,jfog,t)
c
c     Class II subroutine called from subroutines cleanup and chmint.
c
c     Converts equivalent potential temperature to absolute temperature,
c     based on equations (2.7) and (2.8).  Since absolute temperature is
c     an implicit variable in these equations, this subroutine is based
c     on a two-dimensional interpolation of tabulated solutions to
c     these equations; i.e. T = T(tbase, pbase), where
c           tbase = base values of potential temperature function:
c                   th*(p/1000000.)**xk+alpha*rvc
c           pbase = base values of pressure
c           base  = array of tabulated absolute temperatures
c
c     in subroutine argument list:
c
c      p    pressure, dynes/cm2
c      th   equivalent potential temperature in degrees kelvin
c      rvc  mixing ratio of vapor + cloud water
c      jfog flag indicating saturation conditions:
c           1 if saturated; 0 otherwise
```

Table 6.51, Continued

```
c      t    temperature in degrees kelvin
c
       dimension pbase(11),tbase(36),base(11,36)
       data alpha/1725./
       data xk/0.286/
       data (tbase(i),i=1,36)/
     +  0.2300000e+03, 0.2350000e+03, 0.2400000e+03, 0.2450000e+03,
     +  0.2500000e+03, 0.2550000e+03, 0.2600000e+03, 0.2650000e+03,
     +  0.2700000e+03, 0.2750000e+03, 0.2800000e+03, 0.2850000e+03,
     +  0.2900000e+03, 0.2950000e+03, 0.3000000e+03, 0.3050000e+03,
     +  0.3100000e+03, 0.3150000e+03, 0.3200000e+03, 0.3250000e+03,
     +  0.3300000e+03, 0.3350000e+03, 0.3400000e+03, 0.3450000e+03,
     +  0.3500000e+03, 0.3550000e+03, 0.3600000e+03, 0.3650000e+03,
     +  0.3700000e+03, 0.3750000e+03, 0.3800000e+03, 0.3850000e+03,
     +  0.3900000e+03, 0.3950000e+03, 0.4000000e+03, 0.4050000e+03/
       data (pbase(i), i=1,11)/
     +   0.10000e+06, 0.20000e+06, 0.30000e+06, 0.40000e+06,
     +   0.50000e+06, 0.60000e+06, 0.70000e+06, 0.80000e+06,
     +   0.90000e+06, 0.10000e+07, 0.11000e+07/
       data (base(1,i), i=1,36)/
     +  0.2281050e+03, 0.2320880e+03, 0.2357452e+03, 0.2390750e+03,
     +  0.2420810e+03, 0.2447860e+03, 0.2472210e+03, 0.2494200e+03,
     +  0.2514140e+03, 0.2532313e+03, 0.2548961e+03, 0.2564290e+03,
     +  0.2578470e+03, 0.2591650e+03, 0.2603941e+03, 0.2615460e+03,
     +  0.2626290e+03, 0.2636500e+03, 0.2646152e+03, 0.2655312e+03,
     +  0.2664020e+03, 0.2672320e+03, 0.2680250e+03, 0.2687830e+03,
     +  0.2695100e+03, 0.2702080e+03, 0.2708791e+03, 0.2715260e+03,
     +  0.2721493e+03, 0.2727520e+03, 0.2733340e+03, 0.2738973e+03,
     +  0.2744431e+03, 0.2749730e+03, 0.2754870e+03, 0.2759864e+03/
       data (base(2,i), i=1,36)/
     +  0.2289600e+03, 0.2333390e+03, 0.2374710e+03, 0.2413184e+03,
     +  0.2448583e+03, 0.2480880e+03, 0.2510220e+03, 0.2536840e+03,
     +  0.2561020e+03, 0.2583043e+03, 0.2603182e+03, 0.2621671e+03,
     +  0.2638720e+03, 0.2654510e+03, 0.2669181e+03, 0.2682880e+03,
     +  0.2695710e+03, 0.2707770e+03, 0.2719134e+03, 0.2729882e+03,
     +  0.2740070e+03, 0.2749754e+03, 0.2758980e+03, 0.2767782e+03,
     +  0.2776203e+03, 0.2784273e+03, 0.2792020e+03, 0.2799461e+03,
     +  0.2806630e+03, 0.2813540e+03, 0.2820211e+03, 0.2826660e+03,
     +  0.2832894e+03, 0.2838940e+03, 0.2844800e+03, 0.2850483e+03/
       data (base(3,i), i=1,36)/
     +  0.2292820e+03, 0.2338340e+03, 0.2381894e+03, 0.2423040e+03,
     +  0.2461410e+03, 0.2496832e+03, 0.2529301e+03, 0.2558950e+03,
```

Table 6.51, Continued

```
     +  0.2585980e+03, 0.2610644e+03, 0.2633210e+03, 0.2653910e+03,
     +  0.2672980e+03, 0.2690600e+03, 0.2706951e+03, 0.2722183e+03,
     +  0.2736420e+03, 0.2749780e+03, 0.2762341e+03, 0.2774200e+03,
     +  0.2785421e+03, 0.2796070e+03, 0.2806193e+03, 0.2815850e+03,
     +  0.2825062e+03, 0.2833883e+03, 0.2842340e+03, 0.2850460e+03,
     +  0.2858270e+03, 0.2865790e+03, 0.2873041e+03, 0.2880050e+03,
     +  0.2886820e+03, 0.2893372e+03, 0.2899722e+03, 0.2905883e+03/
       data (base(4,i), i=1,36)/
     +  0.2294512e+03, 0.2341010e+03, 0.2385880e+03, 0.2428670e+03,
     +  0.2468980e+03, 0.2506540e+03, 0.2541240e+03, 0.2573111e+03,
     +  0.2602300e+03, 0.2629000e+03, 0.2653453e+03, 0.2675900e+03,
     +  0.2696561e+03, 0.2715650e+03, 0.2733340e+03, 0.2749794e+03,
     +  0.2765160e+03, 0.2779550e+03, 0.2793070e+03, 0.2805811e+03,
     +  0.2817860e+03, 0.2829270e+03, 0.2840111e+03, 0.2850440e+03,
     +  0.2860290e+03, 0.2869703e+03, 0.2878724e+03, 0.2887380e+03,
     +  0.2895700e+03, 0.2903700e+03, 0.2911420e+03, 0.2918861e+03,
     +  0.2926060e+03, 0.2933020e+03, 0.2939760e+03, 0.2946292e+03/
       data (base(5,i), i=1,36)/
     +  0.2295560e+03, 0.2342682e+03, 0.2388420e+03, 0.2432334e+03,
     +  0.2474020e+03, 0.2513150e+03, 0.2549543e+03, 0.2583160e+03,
     +  0.2614060e+03, 0.2642420e+03, 0.2668431e+03, 0.2692330e+03,
     +  0.2714330e+03, 0.2734642e+03, 0.2753461e+03, 0.2770953e+03,
     +  0.2787270e+03, 0.2802534e+03, 0.2816870e+03, 0.2830360e+03,
     +  0.2843100e+03, 0.2855162e+03, 0.2866613e+03, 0.2877510e+03,
     +  0.2887891e+03, 0.2897813e+03, 0.2907310e+03, 0.2916420e+03,
     +  0.2925164e+03, 0.2933580e+03, 0.2941684e+03, 0.2949503e+03,
     +  0.2957060e+03, 0.2964360e+03, 0.2971430e+03, 0.2978280e+03/
       data (base(6,i), i=1,36)/
     +  0.2296271e+03, 0.2343830e+03, 0.2390180e+03, 0.2434920e+03,
     +  0.2477630e+03, 0.2517970e+03, 0.2555701e+03, 0.2590723e+03,
     +  0.2623051e+03, 0.2652800e+03, 0.2680140e+03, 0.2705280e+03,
     +  0.2728440e+03, 0.2749822e+03, 0.2769630e+03, 0.2788022e+03,
     +  0.2805170e+03, 0.2821204e+03, 0.2836244e+03, 0.2850400e+03,
     +  0.2863750e+03, 0.2876380e+03, 0.2888363e+03, 0.2899760e+03,
     +  0.2910610e+03, 0.2920980e+03, 0.2930891e+03, 0.2940400e+03,
     +  0.2949520e+03, 0.2958291e+03, 0.2966740e+03, 0.2974890e+03,
     +  0.2982750e+03, 0.2990352e+03, 0.2997710e+03, 0.3004840e+03/
       data (base(7,i), i=1,36)/
     +  0.2296790e+03, 0.2344670e+03, 0.2391474e+03, 0.2436840e+03,
     +  0.2480350e+03, 0.2521650e+03, 0.2560471e+03, 0.2596670e+03,
     +  0.2630200e+03, 0.2661140e+03, 0.2689631e+03, 0.2715870e+03,
     +  0.2740050e+03, 0.2762380e+03, 0.2783060e+03, 0.2802270e+03,
     +  0.2820160e+03, 0.2836881e+03, 0.2852560e+03, 0.2867300e+03,
```

Table 6.51, Continued

```
     +  0.2881200e+03, 0.2894344e+03, 0.2906804e+03, 0.2918644e+03,
     +  0.2929921e+03, 0.2940683e+03, 0.2950980e+03, 0.2960834e+03,
     +  0.2970300e+03, 0.2979390e+03, 0.2988150e+03, 0.2996583e+03,
     + 0.3004730e+03, 0.3012600e+03, 0.3020220e+03, 0.3027600e+03/
       data (base(8,i), i=1,36)/
     +  0.2297180e+03, 0.2345301e+03, 0.2392470e+03, 0.2438320e+03,
     +  0.2482470e+03, 0.2524560e+03, 0.2564290e+03, 0.2601472e+03,
     +  0.2636041e+03, 0.2668024e+03, 0.2697540e+03, 0.2724750e+03,
     +  0.2749843e+03, 0.2773031e+03, 0.2794504e+03, 0.2814443e+03,
     +  0.2833014e+03, 0.2850362e+03, 0.2866620e+03, 0.2881900e+03,
     +  0.2896300e+03, 0.2909910e+03, 0.2922810e+03, 0.2935054e+03,
     +  0.2946720e+03, 0.2957841e+03, 0.2968480e+03, 0.2978660e+03,
     +  0.2988430e+03, 0.2997820e+03, 0.3006851e+03, 0.3015560e+03,
     +  0.3023960e+03, 0.3032074e+03, 0.3039930e+03, 0.3047530e+03/
       data (base(9,i), i=1,36)/
     +  0.2297481e+03, 0.2345802e+03, 0.2393253e+03, 0.2439502e+03,
     +  0.2484180e+03, 0.2526920e+03, 0.2567413e+03, 0.2605452e+03,
     +  0.2640930e+03, 0.2673830e+03, 0.2704252e+03, 0.2732340e+03,
     +  0.2758270e+03, 0.2782240e+03, 0.2804434e+03, 0.2825050e+03,
     +  0.2844240e+03, 0.2862164e+03, 0.2878954e+03, 0.2894730e+03,
     +  0.2909591e+03, 0.2923632e+03, 0.2936931e+03, 0.2949560e+03,
     +  0.2961574e+03, 0.2973034e+03, 0.2983984e+03, 0.2994470e+03,
     +  0.3004522e+03, 0.3014180e+03, 0.3023472e+03, 0.3032430e+03,
     +  0.3041061e+03, 0.3049403e+03, 0.3057471e+03, 0.3065280e+03/
       data (base(10,i), i=1,36)/
     +  0.2297730e+03, 0.2346210e+03, 0.2393890e+03, 0.2440470e+03,
     +  0.2485580e+03, 0.2528870e+03, 0.2570030e+03, 0.2608810e+03,
     +  0.2645080e+03, 0.2678810e+03, 0.2710050e+03, 0.2738931e+03,
     +  0.2765621e+03, 0.2790310e+03, 0.2813174e+03, 0.2834410e+03,
     +  0.2854180e+03, 0.2872640e+03, 0.2889930e+03, 0.2906162e+03,
     +  0.2921454e+03, 0.2935900e+03, 0.2949570e+03, 0.2962550e+03,
     +  0.2974900e+03, 0.2986670e+03, 0.2997912e+03, 0.3008680e+03,
     +  0.3019000e+03, 0.3028904e+03, 0.3038440e+03, 0.3047620e+03,
     +  0.3056473e+03, 0.3065024e+03, 0.3073293e+03, 0.3081300e+03/
       data (base(11,i), i=1,36)/
     +  0.2297930e+03, 0.2346540e+03, 0.2394420e+03, 0.2441270e+03,
     +  0.2486750e+03, 0.2530520e+03, 0.2572242e+03, 0.2611680e+03,
     +  0.2648660e+03, 0.2683130e+03, 0.2715114e+03, 0.2744730e+03,
     +  0.2772120e+03, 0.2797470e+03, 0.2820960e+03, 0.2842771e+03,
     +  0.2863080e+03, 0.2882041e+03, 0.2899793e+03, 0.2916462e+03,
     +  0.2932160e+03, 0.2946974e+03, 0.2961000e+03, 0.2974310e+03,
     +  0.2986970e+03, 0.2999031e+03, 0.3010552e+03, 0.3021580e+03,
```

Table 6.51, Continued

```
     +  0.3032150e+03, 0.3042292e+03, 0.3052050e+03, 0.3061444e+03,
     +  0.3070504e+03, 0.3079251e+03, 0.3087710e+03, 0.3095890e+03/
c
c      Check and compute for dry air conditions.
       tgroup=th*(p/1000000.)**xk+alpha*rvc
c      assume zero cloud water for low-temperature conditions
       if(jfog.le.0.or.tgroup.lt.230..or.p.lt.3.e5) then
         t=th*(p/1000000.)**xk
         return
       end if
c      check for out of bounds
       if(p.lt.1.e5.or.p.ge.1.1e6.or.tgroup.gt.405.) go to 10
c      locate center points for interpolation
       icntr=max(nint((p-1.e5)/1.e5),1)+1
       icntr=min(icntr,10)
       ic=nint((p-pbase(icntr))/(pbase(icntr+1)-p+.001))
       if(ic.gt.0.and.icntr.lt.10) icntr=icntr+1
       pp=(p-pbase(icntr))/1.e5
       jcntr=max(nint(.2*(tgroup-230.)),1)+1
       jcntr=min(jcntr,35)
       ic=nint((tgroup-tbase(jcntr))/
     +  (tbase(jcntr+1)-tgroup+.001))
       if(ic.gt.0.and.jcntr.lt.35) jcntr=jcntr+1
       qq=(tgroup-tbase(jcntr))/5.
c
c      Now interpolate: use bivariate six-point formula 25.2.67 from
c      Abramowitz and Stegun, Handbook of Mathematical Functions.  NBS
c     Applied Math Series NO. 55, p 882.
c
       t=qq*(qq-1.)*.5*base(icntr,jcntr-1)+
     + pp*(pp-1)*.5*base(icntr-1,jcntr)+
     + (1.+pp*qq-pp*pp-qq*qq)*base(icntr,jcntr)+
     + .5*(pp*(pp-2.*qq+1.))*base(icntr+1,jcntr)+
     + .5*(qq*(qq-2.*pp+1.))*base(icntr,jcntr+1)+
     + qq*pp*base(icntr+1,jcntr+1)
       return
 10    write(9,1000) tgroup,p
 1000  format(1h ,'argument out of bounds'/
     +  'tgroup, p = ',2e12.5)
       t=-999.
       return
       end
```

Table 6.52: Listing of Subroutine **integrate**

```
       subroutine integrate(f,x,ngrid,rturn)
c
c      Class III subroutine called from subroutine matbal
c
c     Variable step-size utility integration routine:
c     Applies a 3-point multisetep quadrature over the domain between n=1 and
c     n=ngrid.  Final step is trapezoidal if ngrid is even.  Accuracy is
c     usually imprved markedly if ngrid is odd.
c
       dimension f(100),x(100)
       nodd=2*ngrid-2*(ngrid/2)-1  !=ngrid if ngrid odd;  = ngrid-1 if even
       rturn=0.
       do 10 n=1,nodd-2,2
         f2=f(n+1)-f(n)
         f3=f(n+2)-f(n+1)
         x3=x(n+2)-x(n+1)
         x2=x(n+1)-x(n)
         x31=x(n+2)-x(n)
         x31sq=x(n+2)**2-x(n)**2
         x31cb=x(n+2)**3-x(n)**3
         firstd=.5*(f2/x2+f3/x3)
         secnd=2.*(f3/x3-f2/x2)/x31
         term0=f(n+1)*x31
         term1=firstd*(.5*x31sq-(x(n+2)-x(n))*x(n+1))
         term2=secnd*(.5*(x31*x(n+1)**2-x31sq*x(n+1))+
     + x31cb*.1666666666667)
         rturn=rturn+term0+term1+term2
 10    continue
c
c      Do trapezoidal on last increment if ngrid is even.
c
       if(nodd.lt.ngrid)
     +   rturn=rturn+.5*(f(ngrid)+f(ngrid-1))*
     +   (x(ngrid)-x(ngrid-1))
       return
       end
```

Table 6.53: Listing of Subroutine **tridag**

```
       subroutine tridag(if,l,a,b,c,d,v)
c
c      Class II subroutine called from subroutines xinteg, yinteg, and sinteg
c
c     Subroutine for solving a system of linear simultaneous equations having
c     a tridiagonal coefficient matrix. The equations are numbered from if
c     through l, and their sub-diagonal, diagonal, and super-diagonal
c     coefficients are stored in the arrays a, b, and c.  The computed
c     solution vector v(if)...v(l) is stored in the array v.
c
        dimension a(l),b(l),c(l),d(l),v(l),
     +  beta(101),gamma(101)
      if(l.gt.101) then
        write(6,*) "l = ",l,
     +        " too large for current dimensioning in tridag"
        stop
      end if
c
c     Compute intermediate arrays beta and gamma.
c
       beta(if) = b(if)
       gamma(if) = d(if)/beta(if)
       ifp1 = if+1
       do 10 i = ifp1,l
         beta(i) = b(i)-a(i)*c(i-1)/beta(i-1)
         gamma(i) = (d(i)-a(i)*gamma(i-1))/beta(i)
 10    continue
c
c     Compute final solution vector v.
c
       v(l) = gamma(l)
       last = l-if
       do 20 k = 1,last
         i = l-k
         v(i) = gamma(i)-c(i)*v(i+1)/beta(i)
 20    continue
       return
       end
```

Table 6.54: Listing of Subroutine **sbstep**

```
       subroutine sbstep(dtsub,nsubt,i,j)
c
c      Class II subroutine called from subroutine core
c
c     Subdivides dt time step into nsubt substeps of length dtsub for
c     integration in the s direction.  nsubt must be an integer.
c
       include 'pluvius.com'
       data courmx /.5/
       dtnew=1.e10
       do 10 k=1,ktot
         wmax=1.e-10
         do 20 ll= 1,lpstot
           wmax=amax1(wmax, abs(windsclm(k)+vertcclm(k,ll)))
 20      continue
         klo=max(k-1,1)
         khi=min(k,ktot-1)
         dzmin=amin1(abs(deltas(klo)), abs(deltas(khi)))
         dtadv=dzmin*courmx/wmax
         dtnew=amin1(dtnew,dtadv)
10     continue
       dtnew=courmx*dtnew
       nsubt = dt/dtnew+1
       dtsub = dt/nsubt
       return
       end
```

Table 6.55: Listing of Subroutine **ijpack**

```
c-----------------------------------------------------------------
       subroutine ijpack( i, j )
c
c     Class II utility subroutine, called from subroutine core.
c
c     Moves variables from i-j-k-l arrays to k-l working arrays for use in
c     z-chem-z integration.  This is just to reduce page faulting.
c
       include 'pluvius.com'
c
c     Pack r's.  After first call (i=j=1), unpack old values then pack new.
```

Table 6.55, Continued

```
c
       if ((i.eq.1) .and. (j.eq.1)) then
         do 10 l = 1, ltot2
           do 10 k = 1, ktot
             rclm(k,l) = r(i,j,k,l)
 10      continue
       else
         do 20 l = 1, ltot2
           do 20 k = 1, ktot
             r(iold,jold,k,l) = rclm(k,l)
             rclm(k,l) = r(i,j,k,l)
 20      continue
       end if
c
c     Pack other variables.
c
       do 30 k = 1, ktot
         windsclm(k) = winds(i,j,k)
 30    continue
       do 40 k = 1, ktot
         do 40 ll = 1,lpsmax
           difusclm(k,ll) = diffus(i,j,k,ll)
 40    continue
       do 50 k = 1, ktot
         cairclm(k) = cair(i,j,k)
 50    continue

       do 60 ll = 1, lpsmax
         do 60 k = 1, ktot
           vertcclm(k,ll) = vertcl(i,j,k,ll)
 60    continue
       do 70 ll = 1, lpsmax
         flxlsclm(ll) = flxls(i,j,ll)
 70    continue
       do 80 ll = 1, lpsmax
         flxhsclm(ll) = flxhs(i,j,ll)
 80    continue
       iold = i
       jold = j
       return
       end
```

Table 6.56: Listing of Subroutine **`ijunpack`**

```
       subroutine ijunpack
c
c     Class II utility subroutine
c
c     Works with ijpack; this routine justs unpack the last column of r's.
c
       include 'pluvius.com'
       do 10 l = 1, ltot2
       do 10 k = 1, ktot
           r(itot,jtot,k,l) = rclm(k,l)
 10    continue
       return
       end
```

Table 6.57: Subroutine Summary and Location Chart

Subroutine	**Class**	**Page Number**
aircon	II	6-148
bndfilt	II	6-120
bounds	I	6-92
boundx	II	6-104
boundy	II	6-113
chmint	II	6-124
cleanup (Case Example A)	I	6-74
cleanup (Case Example B)	I	6-76
cloudinit	III	6-80
coefss	II	6-90
coefsx	II	6-101
coefsy	II	6-110
collect	II	6-77
core	II	6-72
coreOO	II	6-83
diff	I	6-71
esat	II	6-148
esati	II	6-149
femset	II	6-54
foresfilt	II	6-121
gen (Case Example A)	I	6-134
gen (Case Example B)	I	6-138
grdxyz	II	6-84
henry	III	6-151
ijpack	II	6-159

Table 6.57, Continued

ijunpack	II	6-161
inflowinit	I	6-82
init (Case Example A)	I	6-57
init (Case Example B)	I	6-61
inptgn (Case Example A)	I	6-44
inptgn (Case Example B)	I	6-48
integrate	II	6-157
loads	II	6-93
loadx	II	6-105
loady	II	6-114
matbal	I	6-67
odeint	II	6-125
pluvius	II	6-27
print	I	6-67
printi	I	6-66
propss	II	6-88
propsx	II	6-100
propsy	II	6-108
restart	II	6-78
rlapse	II	6-150
sat	II	6-149
ssat	II	6-150
sbstep	II	6-159
sfilter	II	6-116
sinteg	II	6-86
tfrmth	II	6-152

Table 6.57, Continued

`transport`	I	6-95
`tridag`	II	6-158
`wind`	I	6-63
`xfilter`	II	6-117
`xinteg`	II	6-98
`yfilter`	II	6-119
`yinteg`	II	6-107