User's Guide to Parssim1: The Parallel Subsurface Simulator, Single Phase

Version: Parssim1 v. 2.1
May 1998
(Minor updates to February 21, 2006)

Todd Arbogast

Code Management:
The Center for Subsurface Modeling

Mary F. Wheeler, director, Steven Bryant, associate director,
Todd Arbogast, and Clint Dawson

Texas Institute for Computational and Applied Mathematics
The University of Texas at Austin
Austin, Texas 78712

mfw@ticam.utexas.edu, sbryant@ticam.utexas.edu,
arbogast@ticam.utexas.edu, clint@ticam.utexas.edu

Table of Contents

Chapter 1

Brief Summary

Parssim1 is a code developed by the Center for Subsurface Modeling (CSM) of the Texas Institute for Computational and Applied Mathematics (TICAM) at The University of Texas at Austin, Austin, Texas. It is an aquifer or reservoir simulator for the incompressible, single phase flow and reactive transport of subsurface fluids through a heterogeneous porous medium of somewhat irregular geometry. It is also capable of simulating the decay of radioactive tracers or contaminants in the subsurface, linear adsorption, wells and leaking sources, and bioremediation.

Although the code uses very simple rectangular data structures for efficiency and accuracy, the subsurface domain can be of irregular geometry. The subsurface domain is assumed to be described by a logically rectangular grid. A mapping technique is used to map the irregular, physical domain (with its hills, valleys, internal faults and strata, etc.) to the rectangular, computational domain, without loss of accuracy or efficiency. The grid may cover a domain that is periodic in one or more coordinate directions (i.e., periodic boundary conditions are allowed).

The code can run in serial on a single processor, or on a massively parallel, distributed memory computer or collection of computers (using MPI); computational results indicate that it has very good parallel scaling properties. We use *domain decomposition* to compute in parallel. The grid is divided into subdomains, one for each parallel processor. Each subdomain is given roughly the same number of cells. Each processor is responsible for the simulation only in its subdomain. The individual processors send information to each other during the computation.

The program consists of the four main parts: *driver*, *flow*, *transport*, and *chemistry*. The driver routines are responsible for the user interface (input and output) as well as managing the coupling between the flow and transport routines. Chemistry is called from within the transport routines.

The flow is simulated with a package called *Parcel* [14]. It allows for the simulation of incompressible, single-phase, saturated flow with wells on geometrically general domains (but logically rectangular), and it uses a locally conservative, cell-centered finite difference scheme.

The transport routine *ParTrans* allows the simulation of multiphase transport with linear sorption, radionuclide decay, simple (specialized) chemical reactions, and wells on general geometry. Transport is simulated using a locally conservative method of characteristics called the Characteristics-Mixed Finite Element Method (CMM) or a Godunov Method.

The general chemistry routine handles both equilibrium and kinetic reactions. For equilibrium reactions, it uses an interior-point algorithm for the minimization of the Gibbs free energy, and is therefore relatively robust, even when mineral phases precipitate into existence or dissolve away.

Output files can be written in the format used by the visualization tool *Eye* or *Tecplot*, or raw data can be written.

Chapter 2

Governing Equations and Numerical Methods

We describe briefly the flow, transport, chemistry, and radionuclide decay portions of the simulation. Each is solved independently of the others by making use of time splitting techniques [16, 17], as described later in this chapter. Some general references to our approach can be found in [9, 29], and, in a multi-phase setting in [7, 1, 6]. See also the general reference [22].

2.1 Notation

We use the following standard symbolic notation throughout this manual.

c_i	is the molar concentration of species i given in moles per pore volume (i.e., flowing phase volume) (`conc`).
$\mathbf{D}$	is the diffusion/dispersion tensor.
g	is the gravitational constant (`gravity`).
$\mathbf{k}$	is the absolute permeability (`perm`).
p	is the pressure.
q	is a source or sink (i.e., it represents the wells and leaking sources).
t	is time.
$\mathbf{u}$	is the Darcy velocity (`velX`, `velY`, `velZ`).
w_i	is the molecular weight of species i (`molecularWeight`).
x	is the first logical coordinate direction.
y	is the second logical coordinate direction.
z	is the third logical coordinate direction.
α_i	is the sorption coefficient (essentially the Henry's Law constant) (`phaseDist`).
Δt	is the current time step size.
μ	is the flowing phase viscosity.
μ_0	is the viscosity of the resident fluid (`fluidViscosity`).
ν	is the outward unit normal vector to the domain.
ρ	is the flowing phase mass density (`fluidDensity`).
σ_i	is the sorption capacity (`sorp`).
ψ	is the pressure potential (`pressure`), defined as $\psi = p - \rho g z$.

2.2 Flow

The flow equation represents conservation of mass in incompressible single-phase flow, and it is

$$\nabla \cdot \mathbf{u} = q,$$

where the Darcy velocity is

$$\mathbf{u} = -\frac{\mathbf{k}}{\mu}\nabla\psi,$$

and where q is related to `wellInj_f`. The viscosity μ is either μ_0, or it is given as a function of the species concentrations by some mixing rule. This allows simulation of miscible displacement.

2.2.1 Viscosity Quarter Power Mixing Rule

This rule (see [24]) is

$$\mu(\xi_1, ...\xi_n) = \mu_0\Big(\sum_{i=0}^{n} r_i^{-1/4}\xi_i\Big)^{-4},$$

where μ is the viscosity, μ_0 is the resident fluid viscosity of species 0 (`fluidViscosity`), r_i is the ratio of the viscosity of the i species to the resident fluid viscosity (reciprocal of the mobility ratio), and ξ_i is the mass fraction of the ith species. The mass fraction is

$$\xi_i = c_i w_i/\rho.$$

The resident mass fraction ξ_0 is given by solving

$$\sum_{i=0}^{n}\xi_i = 1.$$

We remark that the model should be valid whether or not the resident fluid is one of the species transported in the simulation.

We note that the fluid is assumed incompressible, so the density of the fluid is not affected by its composition (i.e., we assume dilute solutions). This can cause inconsistencies if heavy or light components are present in the fluid in high concentrations.

2.2.2 Numerical Solution

Flow is computed independently of transport, with independent time step sizes. Generally speaking, for time level $m+1$, we solve implicitly

$$\nabla\cdot\mathbf{u}^{m+1} = q^{m+1}, \qquad \mathbf{u}^{m+1} = -\frac{\mathbf{k}}{\mu^m}\nabla\psi^{m+1}.$$

A logically rectangular cell-centered finite difference procedure is used to discretize the equations [14]. This method handles tensor $\mathbf{k}$ accurately [13] and non-rectangular geometry by a mapping technique [4] (see also [11, 8, 12, 3]).

The Glowinski-Wheeler [20] domain decomposition solution procedure is used to solve the resulting equations in parallel. This involves solving an interface problem. The subdomain linear system is solved directly.

2.3 Transport

The transport equation represents conservation of moles of a given species i. For mobile species

$$(\phi+\alpha_i\sigma_i)\frac{\partial c_i}{\partial t} + \nabla\cdot(c_i\mathbf{u} - \mathbf{D}\nabla c_i) = \phi R_i^C(c_1, ..., c_n) + (\phi+\alpha_i\sigma_i)R_i^N(c_1, ..., c_n) + q_{c,i},$$

where R_i^C represents the chemical reactions, R_i^N the radionuclide decay terms, α_i and σ_i are the linear sorption terms `phaseDist` and `sorp`, described below, and $q_{c,i}$ represents the wells as

$$q_{c,i} = q^+ \hat{c}_i + q^- c_i,$$

where q^+ is the positive part of q (i.e., an injection well), q^- is the negative part of q (i.e., an extraction well), and $\hat{c}_i$ is the injected concentration of the well (`wellInj_t`). Leaking sources add a specified amount of moles to specified cells per unit time, but they do *not* affect the flow field. It should be noted that if the general chemistry package is used, as described below, the coefficient of $R_i^C(c_1, ..., c_n)$ is actually $(\phi + \alpha_i \sigma_i)$; thus, care must be exercised when using both the chemistry package and the linear sorption terms–it is recommended that at most one of these be used at a time.

For immobile components, we have a much simpler equation:

$$\frac{\partial c_i}{\partial t} = R_i^C(c_1, ..., c_n) + R_i^N(c_1, ..., c_n) + q_{c,i},$$

where the influence of the well is completely local.

Equilibrium reactions are also supported as the infinite limit of increasing forward and backward rate constants (that maintain a fixed ratio), and subject to local mass balance constraints. This is actually solved by a time splitting technique, so the transport terms and the reaction terms are treated independently of each other. In the limit, equilibrium reactions amount to algebraic constraints on the concentrations, and they are computed *before* or *after* transporting and kinetically reacting all of the species.

2.3.1 Diffusion/Dispersion Tensor

The diffusion/dispersion tensor is

$$\mathbf{D}(\mathbf{u}) = \phi\, d_{\text{mol}} \mathbf{I} + |\mathbf{u}| \left\{ d_{\text{long}} \mathbf{E}(\mathbf{u}) + d_{\text{trans}} (\mathbf{I} - \mathbf{E}(\mathbf{u})) \right\}$$

where $\mathbf{I}$ is the identity tensor, $\mathbf{E}(\mathbf{v})$ is the tensor that projects onto the $\mathbf{v}$ direction and whose (i, j) component is

$$(\mathbf{E}(\mathbf{v}))_{i,j} = \frac{v_i v_j}{|\mathbf{v}|^2},$$

and, if `uniformDispersion` is used, then d_{mol} is `molDiff`, d_{long} is `longDisp`, and d_{trans} is `transDisp`, and, otherwise, d_{mol} is `molDiffAry`, d_{long} is `longDispAry`, and d_{trans} is `transDispAry`.

2.3.2 Linear Sorption Model

A simple linear sorption model is available. It consists of the α and σ terms in the transport equations above. The σ_i vary with space, representing the variation of the rock properties from point to point. The α_i is a relative strength factor for the given species. It is expected that several species may use the same σ_i distribution, and that often $\sigma_i = \phi$.

If linear sorption is used and a species is mobile, the concentration variable `conc` refers to the concentration in the fluid phase. The total concentration T_i is then

$$T_i = (\phi + \alpha_i \sigma_i)\, c_i,$$

so that α_i is essentially the Henry's Law constant. An immobile species has $T_i = \alpha_i \sigma_i c_i$.

Linear and nonlinear sorption can be simulated by the general chemistry routines. If both the linear sorption model described here and the chemistry routines are used, care must be taken in setting up the physical problem. Unlike the general chemistry routines, this linear sorption model does *not* treat the adsorbed or absorbed specie as a distinct specie. It is recommended that only one of these two models be used at a time.

2.3.3 Numerical Solution

The advection and diffusion/dispersion subproblems are solved independently using time splitting. Three options can be selected for the solution of the advection subproblem.

2.3.3.1 Basic Time Splitting Algorithm

The algorithm used can be described by the following steps. We let m denote the current time level and Δt the current time step size. Some of the following sub-problems may not be computed for some or all of the species, as appropriate.

Advection. We solve the equation

$$(\phi + \alpha_i \sigma_i)\frac{\partial c_i}{\partial t} + \nabla \cdot c_i \mathbf{u} = q_{\mathrm{c},i}.$$

For the Characteristics-Mixed Method, we solve for $T_i = (\phi + \alpha_i \sigma_i)c_i$ according to

$$\frac{\partial T_i}{\partial t} + \nabla \cdot \Big(T_i \frac{\mathbf{u}}{\phi + \alpha_i \sigma_i} \Big) = q_{\mathrm{c},i}.$$

by characteristic traking. The result is

$$\frac{\bar{T}_i - T_{i,\mathrm{TB}}^m}{\Delta t} = q_{\mathrm{c},i}.$$

Then $\bar{c}_i = \bar{T}_i/(\phi + \alpha_i \sigma_i)$.

For the Godunov Methods, we solve more directly

$$(\phi + \alpha_i \sigma_i)\frac{\bar{c}_i - c_i^m}{\Delta t} + \nabla \cdot c_i^m \mathbf{u}^m = q_{\mathrm{c},i}.$$

Radionuclide decay. We solve the equation

$$(\phi + \alpha_i \sigma_i)\frac{\partial c_i}{\partial t} = (\phi + \alpha_i \sigma_i) R_i^N(c_1, ..., c_n)$$

explicitly by

$$\frac{\tilde{c}_i - \bar{c}_i}{\Delta t} = R_i^N(\bar{c}_1, ..., \bar{c}_n).$$

Chemistry. We solve for nonequilibrium reactions the equation

$$(\phi + \alpha_i \sigma_i)\frac{\partial c_i}{\partial t} = \phi R_i^C(c_1, ..., c_n)$$

explicitly by

$$\frac{\hat{c}_i - \tilde{c}_i}{\Delta t^*} = R_i^C(\tilde{c}_1, ..., \tilde{c}_n),$$

where $\Delta t^* = \phi \Delta t/(\phi + \alpha_i \sigma_i)$. We then allow for equilibrium reactions satisfying an equation of the form

$$R_i^C(\hat{c}_1, ..., \hat{c}_n) = 0.$$

Diffusion/Dispersion. We solve the equation

$$(\phi + \alpha_i \sigma_i) \frac{\partial c_i}{\partial t} - \nabla \cdot \mathbf{D} \nabla c_i = 0$$

implicitly by

$$(\phi + \alpha_i \sigma_i) \frac{c_i^{m+1} - \hat{c}_i}{\Delta t} - \nabla \cdot \mathbf{D}^m \nabla c_i^{m+1} = 0.$$

2.3.3.2 Characteristics-Mixed Method (CMM)

An explicit characteristics method can be used for advection [2, 10]. The code scales nearly linearly in parallel [5]. No CFL time step constraint is imposed (other than that related to the domain decomposition), and relatively large time-steps can be taken. The scheme has minimal numerical dispersion, but can suffer from numerical mass and/or volume imbalances. It is also relatively computationally expensive.

2.3.3.3 Higher Order Godunov Method (HOG)

An explicit, formally second order Godunov method can be used for advection [15]. A postprocessing step improves the order of accuracy (except near sharp fronts or shocks). The scheme has a CFL time step constraint, so relatively small time-steps must be taken. The scheme has very little numerical dispersion, and is locally mass and volume conservative. Each time step, though small, is computationally relatively inexpensive.

2.3.3.4 First Order Godunov Method (FOG)

The higher order Godunov Method can be used without the postprocessing step, resulting in the first order Godunov method for advection.

2.3.3.5 Diffusion/Dispersion

This subproblem is solved by the cell-centered finite difference technique used for the flow problem, as described above. A simple Jacobi preconditioned conjugate gradient technique is used to solve the resulting linear system.

2.4 Chemistry Overview

Chemistry can be simulated with either user supplied specialized reaction functions, or the general chemistry package [25, 27].

2.4.1 Specialized Chemistry

Specialized chemistry reaction functions can be supplied by the user, compiled into the code, and invoked. The input specification consists almost exclusively of two arrays of real numbers and integers that can be used as parameters in the reaction functions.

The nonequilibrium reactions are solved by a fourth/fifth order Runge-Kutta-Fehlberg technique. The equilibrium functions are assumed to be solved analytically, or iteratively by the user; Parssim1 provides no support for nonlinear solution of these equations. The user supplies the reaction functions R_i either written in C in the file rxn.c or written in Fortran in rxnf.f.

2.4.2 General Chemistry Package

Reactive transport in porous media can be simulated with a full complement of homogeneous and heterogeneous reactions (complexation, adsorption, ion-exchange and precipitation/dissolution) of both equilibrium and kinetic type. Kinetics are assumed to be of the mass-action "distance-from-equilibrium" type, where the reaction rate is proportional to the product of powers of component concentrations.

The package also permits generalised Monod-style rate expressions. This means that many biochemical, biodegradation and bioremediation reactions can be incorporated directly into the general chemistry computation.

The chemistry reactions should be set up as if they were in a batch system; that is, the reaction rates should be given on a moles per unit pore volume (i.e., flowing phase volume) basis. Reactions are not computed in the wells.

Three options exist for the choice of equilibrium module, namely SF (stoichiometric formulation), UNSF (unreduced non-stoichiometric formulation) and RNSF (reduced non-stoichiometric formulation). These options are described in detail in [25]. As a general guideline, we recommend the RNSF version, as it is fastest for most problems that we have encountered. The SF version has the advantage of exact mass-balance and generalizes easier to multi-phase problems, but typically runs slower due to a larger number of variables (in particular, a larger number of Lagrange multipliers). The UNSF is prohibitively slow for realistic simulations and is included mainly for archival reasons. The choice of version is made at the time of compilation.

2.4.3 Numerical Solution

An interior-point optimization technique [19, 26] is used to solve for the minimum of the Gibbs free energy.

2.5 General Chemistry

We reproduce here for convenience information from the chemistry manual [27]. Basic reaction theory is presented. The variables in `typewriter font` refer to variables to be input as data as described in the chapter on the input data file(s) (Chapter 5).

2.5.1 Thermodynamics

In this section we discuss the issues of phase-identity of a species, the usage of practical concentration scales versus computationally convenient ones and unit conversions. The versions that are based on mole-fractions (i.e., SF and UNSF) differ from the molar concentration based version (RNSF), and we discuss these cases separately. In this section, we use notation established by Saaf in his Ph.D. Thesis [25]. In particular, the system is comprised of N_S species (`nSpecies`) of which N_C are components (`nComps`) and N_R products (`nProducts`). When convenient, we distinguish N_R^K kinetic and N_R^Q equilibrium products ($N_R = N_R^K + N_R^Q$). Finally, we let N_M denote the number of minerals and I_M the corresponding index-set.

2.5.1.1 Molar concentration based system (version RNSF)

When the RNSF is used, the composition of the system is computed internally in molar concentrations (c_i, `conc`). The standard mass-action expressions apply; that is,

$$c_{N_C+i} = K_i \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}}, \qquad i = 1, \ldots, N_R, \tag{2.1}$$

with a special case for minerals ($i \in I_M$),

$$\begin{aligned} K_i \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}} &= 1 \quad \text{if } c_{N_C+i} > 0, \\ K_i \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}} &\leq 1 \quad \text{if } c_{N_C+i} = 0, \end{aligned} \tag{2.2}$$

where K_i is the equilibrium constant for the ith reaction and $\hat{a}_{ji}$ are stoichiometric coefficients (`stoich`). In Parssim1, all species participating in `multi-species` phases (including species in the aqueous phase, sorbed phases, or ion-exchange sites) should have `phaseIdentity` = 1, since the chemical potential model is the same for these species (based on molar concentrations). Minerals, however, constitute new phases and should be labeled with `phaseIdentity` $= 2, \ldots, N_M + 1$.

The use of the water component is optional in the RNSF. If water is *not* included, it is an "implicit component" (see e.g. [21]). If it is included, an additional mass-balance equation for water will result. Note that the molar phase density (`chmPhaseDensity`) should be set to a representative value for the aqueous phase (approximately 55.4 mol/liter) if water is included as a component.

2.5.1.2 Mole-fraction based system (versions SF and UNSF)

The versions SF and UNSF are based on internally computing chemical equilibrium in terms of mole-fractions. This treatment is more general than the use of molar variables, but it also carries some limitations. In particular, this formulation allows the simulation of `multi-species` phases other than the aqueous phase (such as a gas phase). This can be accomplished by listing, in additional to the aqueous phase, other multi-species phases in the input file(s). As before, minerals are treated as separate phases and should correspond to different values of `phaseIdentity`.

A difficulty arises in using practical concentration units with a mole-fraction based calculation. To be compatible with the use of molar concentrations in aqueous chemistry, a conversion of the equilibrium data is performed in the code. The conversion modifies the equilibrium constants, based on the assumption of dilute solutions and the specified value of `chmPhaseDensity` for each multi-species phase.

We illustrate this conversion for the simple case of one (aqueous) multi-species phase. The molar concentration of a species, c_l , is defined as

$$c_l \equiv \frac{n_l}{V} = \frac{n_l}{\tilde{n}^{aq}} \frac{\tilde{n}^{aq}}{V}, \tag{2.3}$$

where V is the aqueous phase volume and $\tilde{n}^{aq}$ the total number of moles in the aqueous phase. Introducing the molar phase density

$$\rho^{aq} \equiv \frac{\tilde{n}^{aq}}{V}, \tag{2.4}$$

and using the definition of the mole-fraction of the lth species, we can rewrite

$$c_l = x_l \rho^{aq}. \tag{2.5}$$

Substituting this in the original mass-action expression (2.1) yields

$$x_{N_C+i} = K_i^x \prod_{j=1}^{N_C} (x_j)^{\hat{a}_{ji}}, \quad i = 1, \ldots, N_R, \tag{2.6}$$

where the modified equilibrium constant K_i^x satisfies

$$\log K_i^x = \begin{cases} \log K_i + (\sum_{j=1}^{N_C} \hat{a}_{ji}) \log \rho^{aq} & \text{if } i \text{ is a mineral,} \\ \log K_i + (\sum_{j=1}^{N_C} \hat{a}_{ji} - 1) \log \rho^{aq} & \text{otherwise} \end{cases} \tag{2.7}$$

Similar formulas can be derived if several multi-species phases are present.

As stated above, the conversion assumes dilute solutions. If the actual molar phase density changes drastically, the computed equilibrium will differ from that given by (2.1). Certainly for the aqueous phase the approximation is almost always an excellent one. Note that the solvent (water) must always be present in this formulation, and it is checked that the water component has been specified.

Finally note that, if desired, the user can specify `chmPhaseDensity` = 1.0 for a `multi-species` phase to perform the calculation in mole-fractions (consider e.g. (2.5)). Other units conversions will need to be made carefully elsewhere in the input data file(s) (see §4.7.1 for the units conversion capabilities of the code).

2.5.1.3 Conversion to equivalent reference chemical potentials

When the conditions of chemical equilibrium are formulated as a constrained minimization problem it is the reference chemical potentials $\mu^0 \in \mathbb{R}^{N_S}$, rather than the equilibrium constants, that are the needed in the algorithms. The conversion of the N_R equilibrium constants $K_1, \ldots, K_{N_R}$ into a set of N_S *equivalent reference chemical potentials* is described in [28]. The standard conditions of equilibrium result in the relation

$$K_i = \exp(\frac{-\Delta G_{0,i}}{RT}), \tag{2.8}$$

where the Standard (Gibbs) free energy change of the ith reaction has the form

$$\Delta G_{0,i} \equiv (V^T \mu^0)_i = \mu^0_{N_C+i} - \sum_{j=1}^{N_C} \hat{a}_{ji} \mu^0_j. \tag{2.9}$$

The set of reference potentials corresponding to a given set of equilibrium constants is not unique, and we may without loss of generality assign arbitrary values to N_C of the reference potentials. The most convenient choice is clearly

$$\mu^0_j \equiv 0, \quad j = 1, \ldots, N_C, \tag{2.10}$$

which allows us to write (2.9) as

$$(\Delta G^0)_i = \mu^0_{N_C+i}. \tag{2.11}$$

By combining (2.8) and (2.11) the equivalent chemical reference potentials for the system can be expressed in the simple form

$$\mu^0_{N_C+i} = -RT \log K_i, \quad i = 1, \ldots, N_R. \tag{2.12}$$

2.5.2 Kinetic reactions and their rate-laws

We allow fairly general rate-expressions for the N_R^K kinetic reactions. Expressions derived from classical mass-action equilibrium are supported, as are generalized Monod type kinetic expressions.

2.5.2.1 Distance-from-equilibrium expressions

The distance from equilibrium has long been used as a measure of the driving force for change, in the present application for change in chemical composition. Specifically, we express the kinetic rates r_i^K as the difference between a *forward* and a *backward* rate

$$r_i^K = k_i^f \prod_{j=1}^{N_C} (c_j)^{p_{ji}} - k_i^b c_{N_C+i}, \qquad i = 1, \ldots, N_R^K, \tag{2.13}$$

where k_i^f and k_i^b are the *forward* and *backward* rate constants, respectively, and $p \in \mathbb{R}^{N_C \times N_R^K}$ is a matrix of powers on the components in the rate law. An important special case is the situation $p_{ij} = \hat{a}_{ij}$, i.e., the powers on the component concentrations are equal to their stoichiometric coefficients in the product species. This is the "classical" form of the rate-law. Note that with this choice, the reaction rate is zero if and only if the reaction satisfies the equilibrium conditions with an equivalent equilibrium constant of the form

$$K_i = \frac{k_i^f}{k_i^b}. \tag{2.14}$$

This follows from the fact that

$$\begin{aligned} r_i^K = 0 \quad &\Longleftrightarrow \quad k_i^f \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}} - k_i^b c_{N_C+i} = 0, \\ &\Longleftrightarrow \quad \frac{k_i^f}{k_i^b} \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}} - c_{N_C+i} = 0, \\ &\Longleftrightarrow \quad c_{N_C+i} = K_i \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}}. \end{aligned}$$

The definition of the equilibrium constant for the completed reaction (2.14) allows species-switching (`switchFlag`) to be used (see also §2.5.3.3).

We point out that regardless of the choice of equilibrium module, the rate-laws always use molar concentrations.

2.5.2.2 Monod style expressions

Monod introduced an empirical expression of the form

$$\frac{dX}{dt} = kX \frac{S}{K + S} \tag{2.15}$$

to account for the growth of microbes X on a substrate S. The constant K is commonly referred to as the "half saturation constant" (`halfSatConst`). This expression is widely used to model biologically mediated reactions. We allow a generalized version of this expression:

$$r_i^K = k_i^f \prod_{j=1}^{N_C} (c_j)^{p_{ji}} \prod_{j=1}^{N_C} \frac{c_j}{K_{ij}^{half} + c_j} - k_i^b c_{N_C+i}, \qquad i = 1, \ldots, N_R^M, \tag{2.16}$$

where K_{ij}^{half} are the half saturation constants for the component species (more than one component is allowed to participate in each reaction) and N_R^M is the number of Monod style rate expressions. Any or all or none of the N_R^K kinetic reactions may have Monod-style expressions, i.e., $0 \leq N_R^M \leq N_R^K$. When $K_{ij}^{half} = 0$ for all j the Monod expression reduces to the distance-from-equilibrium expression. Typical bioreactions are irreversible, and the backward rate constant k_i^b is usually set to zero.

2.5.2.3 Kinetic reaction products

It may happen that several kinetic reactions yield a common product or by-product, e.g., carbon dioxide, or that a single reaction produces multiple products. The framework described above makes no explicit accommodation of these possibilities, but it can in fact be extended to these situations. The trick is to define some common products or by-products as components and assign them negative stoichiometric coefficients (`stoichK`) in the reactions for the other product species.

2.5.3 Numerical parameters

We now discuss appropriate settings for the chemistry solution parameters (in §5.2.3). For some of these parameters, useful default values are given there.

The parameter `chmLoadBalFlag` specifies whether automatic parallel load balancing of the computational work among processors should be attempted. This creates parallel communication overhead, but generally speeds up the computation in parallel.

2.5.3.1 Interpretation of the results

Let us begin by considering termination criteria for the equilibrium calculation. The parameter `chmMaxIter` specifies the largest number of iterations allowed to satisfy the stopping criterion; if this is number is insufficient, an error message will result and the calculation fails. Regardless of what version is used, the iterates are required to satisfy

$$\|F\| \leq \texttt{chmAbsTol}, \tag{2.17}$$

i.e., the L_2 norm of the residuals (the KKT conditions [18]) must be less than or equal to the user-specified tolerance. This condition has different implications depending on the choice of equilibrium module; however, and we discuss here the SF and the RNSF.

Using the SF, mass-balance is exactly satisfied, and the residuals measure deviation from *optimality* and *complementarity*, as explained in [25]. It is easy to see that in a system in which all species have concentrations which are sufficiently large, satisfaction of the stopping criterion will approximately guarantee that for the ith reaction

$$|\Delta G_i| \leq \texttt{chmAbsTol}, \tag{2.18}$$

where G_i is the Gibbs free energy. These results can be expressed as

$$\left|\frac{c_{N_C+i}}{K_i \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}}}\right| \leq \exp(\texttt{chmAbsTol}) \approx 1 + \texttt{chmAbsTol}, \tag{2.19}$$

or, equivalently,

$$\frac{|c_{N_C+i} - K_i \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}}|}{K_i \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}}} \leq \texttt{chmAbsTol}, \tag{2.20}$$

which clearly shows the role of `chmAbsTol` as a kind of relative error in this application. For species present at small concentrations, this bound will deteriorate.

In the RNSF, on the other hand, mass-balance is only approximately satisfied by a given iterate, and the absolute value of the mass-balance error for any component is bounded by `chmAbsTol`. The implication is that even a small specified tolerance could result in a large relative error for a component which is present in trace amounts only. Note that in the RNSF the residuals consist entirely of component mass-balances in the absence of minerals. If minerals are present, this set is augmented by one complementarity and one optimality equation for each mineral. It can be shown that mineral concentrations in the computed solution will satisfy the inequality

$$-\texttt{chmAbsTol}(1 + c_{N_C+i}^{-1}) \leq \log(K_i \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}}) \leq \texttt{chmAbsTol}, \quad i \in I_M, \tag{2.21}$$

which is a satisfying result. In particular, if the mineral is present at a large concentration $c_{N_C+i} \approx 1$, this is approximately

$$|K_i \prod_{j=1}^{N_C} (c_j)^{\hat{a}_{ji}} - 1| \leq \texttt{chmAbsTol}. \tag{2.22}$$

For numerical reasons, a non-zero tolerance `chmEpsConc` > 0 is used in enforcing the condition of non-negativity of species. When the SF is used, all species concentrations affected by the equilibration are bounded below by `chmEpsConc`. For the RNSF, which uses logarithmic transformations of some of the variables, this restriction is only in place for mineral species; other species can be arbitrarily close to zero. It is assumed that `chmEpsConc` $\ll$ `chmAbsTol`.

The variable `chmTestSolFlag` enables a test of the second-order sufficiency conditions at the computed solution, i.e., that the Hessian projected onto the tangent space of the active constraints have all positive eigenvalues (see, e.g., [30]). This procedure can only be used with versions UNSF and SF. It is probably of limited value for most simple aqueous ideal systems which have known convexity properties. It could prove useful for batch testing of more complicated multi-phase systems.

2.5.3.2 Algorithmic parameters for the nonlinear iteration

The parameter `chmGuessType` specifies the type of initial guess used to initialize the nonlinear equilibration iteration.

The group of parameters `chmInterpFlag`, `nViol`, and `chmAlpha` relate to the line-search globalization of the nonlinear solver. The merit function is the square of the L_2 norm of the perturbed KKT conditions (see [18, 25, 26] for details).

The well-known Armijo-Goldstein condition (see [18]) is imposed at each step with the setting `nViol` $= 0$; if a value greater than zero is specified, `nViol` violations of this condition are accepted before it is again required to hold. The parameter `chmAlpha` is the fraction of the initial decrease that the decrease resulting from the adjusted step is required to have, and `chmInterpFlag` indicates whether a simple reduction of the step-length or a more costly cubic/quadratic interpolation procedure is used in the line-search step (see [18]).

Often, the algorithm converges significantly faster if a few violations of the Armijo-Goldstein condition are tolerated, and for this reason we recommend values of `nViol` in the range of 1 to 5. However, for highly sensitive problems, it may be necessary to use a value of 0 to avoid jeopardizing global convergence. It is possible (and sometimes useful) to combine the setting `nViol` $= 0$ with `chmAlpha` < 0. This is another way of "relaxing" the Armijo-Goldstein condition by allowing a slight increase in the merit function.

The variables `tauMin` and `chmRho` are directly related to the interior-point algorithms. Variable `tauMin` describes the smallest percentage of the movement to the non-negativity boundary that is allowed. It should be taken close to (but strictly less than) unity.

The parameter `chmRho` is the fraction by which the perturbation parameter in the interior-point method (see [26]) is decreased as the solution is approached. It is always less than unity. Typically, a small value gives the fastest convergence, but too small a value may lead to the global convergence problem of "sticking to the boundary" (see [19]).

Note that the settings of `tauMin` and `chmRho` are of importance only when inequality constraints are present in the formulation. Thus, if the RNSF version is used to solve a problem without mineral phases, these settings are of no consequence.

2.5.3.3 ODE integration control

The parameters `ntReact`, `odeAlgType` and `switchFlag` control the time-integration of the ODEs governing kinetic reactions and are only used if such reactions are specified by `reactionType`.

The variable `odeAlgType` controls the type of explicit time-stepping scheme used for the integration. Currently, the choices 0, 1, 2 are supported, corresponding to the use of Forward-Euler (FE) first-order scheme, second-order Runge-Kutta (RK-2) and fourth-order Runge-Kutta (RK-4).

The parameter `ntReact` defines a target time-step to be used in the kinetic integration. It specifies the smallest number of reaction steps per transport step that we consider adequate. As an example, `ntReact` = 1 means that the integration of the ODEs will proceed with a time-step equal to that used for transport. Internally to the chemistry routines, this time-step size will always be *attempted* at the beginning of a step; however, it may be *cut* during the step if non-negativity violations are encountered.

We point out that the variable `chmEpsConc` plays the role of a user-specified approximate zero concentration in the kinetic integration. We do not allow any kinetic process to decrease a species concentration below `epsConc`.

If `switchFlag` = 0, the sub-division into equilibrium and kinetic reactions will remain the same throughout the simulation. This is the normal mode of operation. However, with the setting `switchFlag` = 1, the algorithms will attempt to "switch" kinetic reactions that appear to have gone to completion to the equilibrium subset. The "equivalent" equilibrium constant defined in (2.14) is used for this purpose. We point out that if species-switching is used, all rate-laws should be in the "classical" form, as noted above; otherwise, the results are unpredictable.

2.6 Radionuclide Decay

Radionuclide decay reactions can be handled by the chemistry routines or by a separate radionuclide decay routine that is supplied. If used, it amounts to a time splitting of the radionuclide decay reactions. These exponential growth and decay reactions can be extremely stiff, and they are solved by a diagonalization of the system to increase stability and robustness. The half-lives of the species *must* be unique.

Chapter 3

Program Name and Command Line Arguments

The executable program for Parssim1 is called simply parssim.

The unix calling specification is:

```
parssim [-i [<file name>]] [-e [<file name>]] [-E]
          [-c [<file name>]] [-t] [-u] [-v] [-h]
```

The optional command line arguments are:

`-i [<file name>]` The initial general input file `<file name>`, or "infile" if omitted.

`-e [<file name>]` Echo input to file `<file name>`, or "echo" if omitted. This is useful to track down errors in the input files. It is also useful to strip most comments out of the data files and merge the included files into a single file, since the echo file is in the form of an input file.

`-E` Echo the input to standard out. This is useful to track down errors in the input files.

`-c [<file name>]` Echo general chemistry input data to file `<file name>`, or "echoChem" if omitted. This is useful to track down errors in the input description of the general chemical reactions (if used).

`-t` Just display Parssim1 input type information. These type numbers are used to input data.

`-u` Just display Parssim1 input physical units information. This allows the user to see the predefined units available, their dimensions, and the initial internal base units used within the code. Note, however, that the base units can be modified by the user.

`-v` Just display Parssim1 version information.

`-h` Print a summary of the above usage information only.

Chapter 4

General Input File Syntax and Physical Units

All input is read from one or more data files. Key-words and data items are arranged in sections, subsections, and sub-subsections and ordered item-by-item, possibly separated by white-space, comments, and list delimiters, and possibly special commands.

4.1 White-space

White-space is used to separate other items, and is otherwise ignored on input. White-space consists of the characters for space, tab, newline (or end-of-line), comma '`,`', semicolon '`;`', and colon '`:`'. White-space is ignored on input. The punctuation characters allow the user to delineate sets of items, for example.

Proper white-space consists of the characters for space, tab, and newline. Command names and key-words are terminated by proper white-space.

4.2 Commands

Commands alter the way the data file(s) are read. Commands take the form command symbol '`$`' followed immediately (no space) by a command name. Note that the command symbol is ignored in any explicit comment and in skipped sectional units, but *not* before the first section (see the discussion below on comments and sectinal units).

A list of commands follows.

`$` Null command.

`$$` Null command.

`${#` Equivalent to `$BEGIN_COMMENT`.

`$#}` Equivalent to `$END_COMMENT`.

`$BEGIN_COMMENT` Causes input to be ignored until the matching `$END_COMMENT` command is read (such commands are nested) or the end-of-file is reached.

`$END_COMMENT` Terminates a `$BEGIN_COMMENT` command.

`$IGNORE_LINES` Follow by proper white-space and an integer. Causes that number of lines (i.e., up to newline) to be ignored from the input stream. The line containing the integer is included in the count of lines ignored, and information is ignored from that point on. However, the include command only is *not* ignored. This command can be used to ignore preambles in included data files.

`$IGNORE_WORDS` Follow by proper white-space and an integer. Causes that number of words to be ignored from the input stream. However, the include command only is *not* ignored. This command can be used to ignore preambles in included data files.

`$INCLUDE` Follow by proper white-space and a new file name (maximum of 50 characters). This causes subsequent reading to take place in the new file. When the end-of-file is reached, reading commences in the original file. While there is no limit on the number of files that may be included, there is a maximum level to which they may be nested.

`$LITERAL` Terminates reading white-space and other symbols. This command is used to allow a special symbol at the beginning of a data string (such as punctuation '`,`', '`;`', and '`:`', comment symbol '`#`', and command symbol '`$`'). (This command is not needed before `$INCLUDE` file names.)

`$S` Equivalent to `$SECTION`.

`$SECTION` Begins a section of the data file. Follow by the section name.

`$SS` Equivalent to `$SUBSECTION`.

`$SSS` Equivalent to `$SUBSUBSECTION`.

`$SUBSECTION` Begins a subsection of the data file. Follow by the subsection name.

`$SUBSUBSECTION` Begins a sub-subsection of the data file. Follow by the sub-subsection name.

4.3 Comments

Explicit comments appear either after the comment symbol '`#`' or between the `$BEGIN_COMMENT` and `$END_COMMENT` commands. A comment begun with '`#`' runs to the newline symbol, so it can be used to remove lines or ends of lines from a data file, without actually losing the information. (Note that '`#`' is an ordinary character if it is embedded in a string). It is acceptable to nest `$BEGIN_COMMENT` and `$END_COMMENT` commands.

Implicit comments can be made before the first section. Note that an unprocessed sectional unit is an implicit comment. Moreover, sectional units given bogus names (such as "remark") can be used also as implicit comments.

4.4 Sectional Units

Sections, subsections, and sub-subsections are initiated by a `$SECTION` command, `$SUBSECTION` command, or `$SUBSUBSECTION` command, respectively, and a specified string. It is important to note that entire sectional units may be skipped on input if they are not needed according to the options specified by the data. This allows major routines to be omitted from the calculation (such as flow, transport, or chemistry) without modifying the input file(s). If there is any doubt, one can use the command line argument `-e` or `-E` to see if a given sectional unit was actually processed.

4.5 List Delimiters

List delimiters enclose lists that have a user specified number of items. A list is begun with a begin list symbol '{' and ended with an end list symbol '}'. The delimiters are used to check that the user has counted the number of list items correctly.

4.6 Key-words

Key-words are specified strings of characters. Spaces may be embedded in a key-word.

4.7 Data Items

A data item can be an integer (not be followed by a period), a real number (in standard floating point or exponential notation), a word of characters (terminated by a space, newline, or tab), or a line of characters (including spaces and tabs, terminated by a newline character).

Data items of characters are limited in length. The maximum allowed number of characters is:

```
120 for a directory name;
 50 for a file name;
120 for a line of character;
 30 for a word of characters;
 15 for phase and species names.
```

4.7.1 Physical Units

Real numbers have physical units. Physical units can be optionally assigned by using the square bracket notation

```
<real> [<units expression>],
```

where the units expression may contain integer and real numbers, the names of units, arithmetic operators, parentheses, and spaces (which are ignored). The statement declares the physical units of the number `<real>`, and causes the value to be converted into the set of base units used internally in the code. The command line argument `-e` or `-E` displays the conversion computation for diagnostic purposes. The base units can be set by the user (see §5.1 below).

A list of supported units, with their physical dimensions (and *initial* base unit numerical values), can be found by using the command line argument `-u`. The arithmetic operations supported are integral exponentiation '`**`' or '`^`', multiplication '`*`', division '`/`', negation '`-`', and parentheses '`(`' and '`)`'. Exponentiation by an integer with no physical dimensions only is allowed. We also allow addition '`+`' and subtraction '`-`', though these should not appear in a units expression. The usual rules of precedence apply: exponentiation is performed first from left to right, then multiplication and division, and finally addition and subtraction, although parentheses can be used to alter this order. A leading '`*`' or '`/`' is allowed (a preceding '`1`' is assumed).

Physical units are optional; the set of internal physical base units are assumed if none are given. If units are given, the physical dimensions must be proper for the given quantity or an error will result. Dimension checking can be disabled by using double square bracket notation, as in

```
<number> [[<units expression>]],
```

in which case addition and subtraction are allowed. This also enables one to include the numerical values of unsupported units.

It is relatively easy for a programmer to add a new unit name to the code (or to change the initial base units used internally in the code).

Temperature degree increments are an ordinary case of units as described above; however, temperature readings are a special case. These need simply the temperature scale used, degrees Kelvin (`degK`), Celsius (`degC`), or Fahrenheit, (`degF`).

Examples follow.

```
2e-5 [cm/sec**2]  # floating point notation is allowed
8.2 [kg*m*sec^-2] # negation and integral exponentiation are allowed
4.37 [[2]]        # dimensions may be incorrect; value is doubled
50 [/yr]          # initial division is allowed
1.2 [[2.1e-4 * (6 + 8.1^(cm/m)) / 2**3 + (cm/sec)^4]]
                  # nonsense, but a correctly formed expression
100 [degC]        # a temperature reading
```

4.7.2 Data Blocks and the Repetition Symbol

Data items may come in a block of data, as when a grid array (see below) is specified. Such a data block is bounded by the list delimiter symbols '{' and '}', as described above. Within such a block, the repetition character '`@`' may be used. A set of "`n`" single data items, each with the value "`value`," may be indicated by

```
n@value.
```

Proper white-space may precede "`value`," but only spaces may precede '`@`'.

Units may be specified for the real numbers in a data block, but only for the list as a whole; that is, only a single expression can be given, and it applies to each number of the data block. The units expression must appear just before the list of numbers, as in

```
{ [cm^3/sec] 2.1 3.2 4@7.5 ...  }.
```

4.7.3 Grid Arrays: Constant, Nearly Constant, and Fully Specified

Data may be distributed in space and therefore assigned to every cell or point of the grid in 3-D, or in 2-D to every face or point of the domain boundary. The data per grid cell may be a single numerical value or a set of numerical values of a given fixed size. A uniform syntax is used to specify such an array.

Within the array, the x coordinate varies most rapidly, then y, and finally z (as in Fortran). For cell-centered data, the index range goes from 1 to the number of grid cells in the given direction (i.e., the full range is `(1:nx,1:ny,1:nz)`). For point-centered data, the index range goes from 0 to the number of grid cells in the given direction (i.e., the full range is `(0:nx,0:ny,0:nz)`). For face-centered data, the index range depends on the type of face considered. For an x-face, the array is point-centered in the x-direction and cell-centered in the other directions; similarly for a y-face and a z-face (i.e., the full range is `(0:nx,1:ny,1:nz)`, `(1:nx,0:ny,1:nz)`, and `(1:nx,1:ny,0:nz)` for x-face, y-face, and z-face face-centered data, respectively.)

A grid array that consists of a single set of repeated values (i.e., a constant set of values) can be specified as

```
constant <value(s)>,
```

where, if the values are real numbers, each may have units. The number of values needed is the number that the array requires per grid cell or point.

A grid array that consists of a few repeated sets of values (i.e., an array that is nearly constant) can be specified easily. For a 3-D array of grid cell or point values, the specification is

```
nearly constant <primary value(s)>
<number not that or those values>
{
```

```
<low x index> : <high x index> , <low y index> : <high y index> ,
  <low z index> : <high z index> ; <value(s)>
...
},
```

where the punctuation is not necessary, since it is merely white-space, the curly brackets are list delimiters that count the number `<number not that or those values>` of exceptional patches, and the number of values needed is the number that the array requires per grid cell or point. The meaning is as follows. Fill the array with the first set of values `<primary value(s)>` given. Next, for the range of indices specified by each set of six integers, overwrite the array with the given set of values `<value(s)>`. Subsequent exceptional patches may overwrite previous patches. For a 2-D array of grid cell or point values on a boundary face, the specification is

```
nearly constant <primary value(s)>
<number not that or those values>
{
<low first index> : <high first index> ,
  <low second index> : <high second index> ; <value(s)>
...
},
```

where the meaning is similar, but now the ranges depend on the face considered (e.g., for an x-face the full cell-centered range is `(1:ny,1:nz)`).

A grid array that is to be fully specified is given as a data block (see above). A data block arising to specify a grid array can require a tremendous amount of information. It may be convenient to put this information in an auxiliary data file, in which case, it could be read easily as

```
{ $INCLUDE <file name> }.
```

It is easy to incorporate the auxiliary data file even if it has the wrong physical units. Moreover, it is easy to ignore any preamble that the file may contain using either the `$IGNORE_WORDS` or `$IGNORE_LINES` commands, as in

```
{ [kg/m^3/hr] $IGNORE_LINES 2 $INCLUDE <file name> }
```

(which ignores the first two lines of the file `<file name>` and applies units conversion to the values).

Examples follow.

```
constant 2.7e-2 [m]

nearly constant 2.7e-2
1
{
3:3,2:2,5:5 ; 2.7e-2 [[1/2]] # cell or point (3,2,5) has half the value
}

nearly constant 4.6, 5.2 # 2 items per cell or point
2
{
1:3,2:4,5:5 ; 2, 7
2:3,2:3,4:5 ; 3.1,6 # overlapping areas have the values 3.1 and 6
```

```
}

nearly constant 2.5 , 1 {1:2,0:3; 6.7} # a boundary grid array

{ 2.5 3@4.7 8@0 3.5e2 ...  } # fully specified
```

Chapter 5
The Input Data File(s)

All input is read from data files that contain information arranged in sections, subsections, and sub-subsections. This data is read item-by-item, and order is important.

Call the initial input file "infile," or use the command line argument `-i` to specify the name of the initial file. Include commands can be used to separate data into additional files if this is convenient or necessary.

Below is an item-by-item description of the input file, divided into its various sections. Manual section headings and indented remarks are not part of the input file(s). To help distinguish the input file text from merely manual text, the `typewriter font` is used for input text.

The previous section describes general syntactical features of the input reading routines that may be exploited quite generally, such as declaring units and including comments; these features will not be repeated below.

Data items are described literally or by type. Literal optional forms of an item are separated by "`||`"; only one option should be selected. Item types names are enclosed in angled brackets, as in `<integer>`. Real numbers have physical units: the dimensions are given as in `<real [M/(LT^2)]>`, where

L refers to length,
M refers to mass,
T refers to time.

In a few cases, a suggested value is given after a colon, as in `<integer: 100>`.

Comments are optional, of course. A boolean is a variable that is the answer to a yes or no question (value 1 or not 0 is yes, value 0 is no). A flag is variable that provides a selection from several options as specified by some integer.

Sample input data files appear in Appendix A. All symbolic notation is defined above in §2.1 on page 6. A template of the input data file is also available; it consists of the appropriate lines (those in `typewriter font`) below.

5.1 General Information

```
$S GENERAL INFO
0 || 1    # computeFlow:  Compute flow
```

Boolean for whether a flow field will be computed. If no flow field is computed, the flow field should be given in §5.12 on initial conditions.

`0 || 1 || 2 || 3    # computeTransport:  Compute transport`

Flag declaring how the transport (advection) will be computed. The command line argument `-t` gives the options:

0 for no transport;
1 for the Characteristics-Mixed Method (CMM);
2 for the higher order Godunov Method (HOG);
3 for the first order Godunov Method (FOG);
-1 for diffusion (and dispersion) only, i.e., diffusion but not transport is computed (it is recommended for consistency that one specify `computeFlow` = 0 above, a zero velocity field in §5.12 on initial conditions, and either noflow conditions in §5.13 on boundary conditions or periodic boundary conditions in §5.4).

`0 || 1 || 2 || 3 || -1    # computeChemistry:  Compute chemistry`

Flag declaring whether chemistry will be computed. The command line argument `-t` gives the options. The full set of options is:

0 for no chemistry;
1 for the general chemistry RNSF option;
2 for the general chemistry SF option;
3 for the general chemistry UNSF option;
-1 for the specialized chemistry routines.

If option 1, 2, or 3 is chosen, the code must have been compiled for that option or an error will result.

`0 || 1    # computeRND: Compute RND`

Boolean to invoke the radionuclide decay routines.

`[<units expression [L]>]    # baseLength:  Internal length base units`

Set the internal length base units. This overrides the initial values that were compiled into the code. The initial default values can be retained by using the value `[[1]]`.

`[<units expression [M]>]    # baseMass:  Internal mass base units`

Set the internal mass base units. This overrides the initial values that were compiled into the code (these can be retained by using the value `[[1]]`).

`[<units expression [T]>]    # baseTime:  Internal time base units`

Set the internal time base units. This overrides the initial values that were compiled into the code (these can be retained by using the value `[[1]]`).

`[degK] || [degC] || [degF]    # baseTemperature:  [degK], [degC], or [degF]`

Set the internal temperature base scale and units. See §4.7.1 for temperature readings. This overrides the initial values that were compiled into the code (these can be retained by using the value `[[1]]`).

5.2 Algorithm Solution Parameters

`$S SOLUTION PARAMETERS`

```
<integer: -1> <integer: -1> <integer: 1>    # nDom_x, _y, _z:
        # Parallel subdomain divisions (-1=auto select)
```

The number of processors or subdomains in each of the three coordinate directions. Note that their product must equal the total number of processors used in the simulation. An automatic selection in any direction can be obtained by specifying the number -1. Note that if there are wells, `nDom_z` *must* be set to 1.

5.2.1 Flow

```
$SS Flow
# This sectional unit is needed only if computeFlow != 0
```

```
<integer: 100>    # maxIterIF_f:  Interface maximum number of iterations
```

Maximum number of iterations for solving the flow interface problem.

```
<real [1]: 1.0e-6>    # relTolIF_f:  Interface relative tolerance
```

Relative tolerance for solving the flow interface problem.

```
<real [1]>    # absTolIF_f:  Interface absolute tolerance
```

Absolute tolerance for solving the flow interface problem. Its value depends on the given problem, since it is unscaled. A very small number or even 0 is appropriate in most cases.

```
0 || 1 || 5 <: 5>     # pcTypeIF_f:  Interface preconditioner
```

Flag to select the preconditioner for solving the flow subdomain interface problem. The command line argument `-t` gives the options:

0 for no preconditioner;
1 for diagonal (Jacobi) preconditioning;
5 for a balancing preconditioner.

```
<integer: 48000>    # dWorkspace_f:  Double workspace for flow
```

The amount of Fortran double precision workspace to allocate for the flow routines. The flow routines will abort if this value is too small. In that case, raise its value and rerun the code. The amount of workspace needed depends on the subdomain grid size.

5.2.2 Transport

```
$SS Transport
# This sectional unit is needed only if computeTransport != 0
```

```
<integer: 100>    # maxIter_t:  Dispersion maximum number of iterations
```

Maximum number of iterations for solving the transport diffusion/dispersion problem.

```
<real [1]: 1.0e-6>    # relTol_t:  Dispersion relative tolerance
```

Relative tolerance for solving the transport diffusion/dispersion problem.

```
<integer: 10000>    # dWorkspace_t:  Double workspace for transport
```

The amount of Fortran double precision workspace to allocate for the transport routines. The transport routines will abort if this value is too small. In that case, raise its value and rerun the code.

```
<real [1]: -1>    # cflFactor:  Numerical CFL factor
```
The numerical CFL constraining factor applies to the advection schemes. If this number is negative, the desired fixed time step size `dtMax_t` will be attempted. If this number is positive, an automatic selection will be made.

If the CMM option is selected by `computeTransport`, then the CFL limit refers to the maximum amount of characteristic tracking that may take place in each subdomain's overlap region (padded or shadow region) specified by `padx`, `pady`, and `padz`. The `cflFactor` will then be used to select a time step that is the given fraction (`cflFactor`) of the maximum estimated time step that would keep each subdomain characteristic track within its subdomain or overlap region.

5.2.2.1 Characteristics-Mixed Method

```
$SSS CMM
# This sectional unit is needed only if computeTransport
# selects the CMM advection scheme.
<integer: 2>    # ntCutMax_t:  Maximum number of time step cuts
```
The number of times the transport time step size can be cut (per step attempted).

```
<integer: 2> <integer: 2> <integer: 2>    # padx, pady, padz:
        # Number of cells for subdomain overlap (pad)
```
The number of extra grid cells in each coordinate direction in the overlap region (padded or shadow region) on each end of each subdomain.

```
<real [1]: 10>    # volTol:  Volume discrepancy tolerance
```
The tolerance for the transport's relative volume discrepancy.

```
<integer: 1> <integer: 1> <integer: 1>
        # volRefine_t_x, volRefine_t_y, volRefine_t_z: Trace-back volume refinement
```
Trace-back volume refinement in each direction. The value 1 is no refinement (full grid cells will be traced back); otherwise, each grid cell can be divided along the coordinate directions into an integral number of pieces, and each sub-cell will be traced.

5.2.3 Chemistry

```
$SS Chemistry
# This sectional unit is needed only if computeChemistry > 0
0 || 1 <: 1>    # chmLoadBalFlag:  Parallel chemistry load balancing
```
Specifies whether automatic parallel load balancing of the computational work among processors should be attempted. This creates parallel communication overhead, but generally speeds up the computation in parallel.

```
<integer: 100>    # chmMaxIter:  Maximum number of nonlinear iterations
```
The maximum number of iterations allowed in the nonlinear solver.

```
<real [1]>    # chmAbsTol:  Absolute KKT tolerance
```
The *absolute* tolerance on the residuals of the KKT conditions (see [18]) used in the stopping criterion, as discussed in §2.5.3. Caution must be exercised with species of small total concentration (see §2.5.3.1).

```
<real [moles/L^3]]: 1.0e-16>    # chmEpsConc:  Minimal concentration parameter
```
A lower bound on concentrations in the minimization algorithm (see §2.5.3 and §2.5.3.1). This munber must be much less than `chmAbsTol`.

`0 || 1 <: 0>  # chmScaleFlag:  Diagonal scaling`
Boolean to attempt improvement in the condition number of the Newton system by a diagonal scaling procedure.

`0 || 1 <: 0>  # chmTestSolFlag:  Test second-order sufficiency conditions`
Boolean to test second-order sufficiency conditions of optimality at the computed solution (see §2.5.3).

`-1 || 0 || 1 <: -1>  # chmGuessType:  initial guess (-1=auto,0=transported,1=previous)`
Flag to select the initial guess for the equilibration computation. The command line argument `-t` gives the options:

0 for the transported concentration values;
1 for the previous concentration values;
-1 for an automatic selection (0 for pure equilibrium, 1 otherwise).

`0 || 1 <: 0>  # chmInterpFlag:  Use interpolation in the back-track line-search`
Boolean to select interpolation in the back-tracking algorithm used in the line-search step; otherwise, use simple reduction (see §2.5.3.2).

`<integer: 10>  # nViol:  Number of non-monotonic line-searches`
A control for the back-track line-search algorithm. If 0, the standard Armijo-Goldstein condition of monotonicity (see [18]) is imposed on the step. If positive, we tolerate that many consecutive violations of this condition before it is imposed again (see §2.5.3.2).

`<real [1]: 1.0e-4>  # chmAlpha:  Alpha parameter`
The alpha parameter in the Armijo-Goldstein condition (see §2.5.3.2 and [18]).

`0 || 1 <: 1>  # hessFlag:  Analytical Hessian`
Boolean to select use of the analytic Hessian. Otherwise, a finite-difference computation of the Hessian is performed (for the SF and UNSF versions). Currently, the finite-difference code is redundant, but included for future use with non-ideal systems. It is best to use the analytic Hessian (option 1).

`<real [1]: 0.8>  # tauMin:  Movement to the boundary factor`
Smallest allowable percentage of movement to the boundary in the interior-point algorithm (see §2.5.3.2 and [25]).

`<real [1]: 1.0e-2>  # chmRho:  IP reduction factor of perturbation parameter`
The multiplicative reduction of the interior-point perturbation parameter (see §2.5.3.2). Must be less than 1.

`<integer>  # ntReact:  Number of reaction steps per transport step`
Target number of reaction sub-time steps per transport time-step. The value chosen depends on the relative time-scales involved in transport and reactions (see §2.5.3.3).

`0 || 1 || 2  # odeAlgType:  ODE integration (0=RK1, 1=RK2, 2=RK4)`
Flag to select first (0), second (1), or fourth-order (2) in time accurate Runge-Kutta integration of kinetic reactions, respectively (see §2.5.3.3).

`0 || 1 <: 0>  # switchFlag:  Species switching`
Boolean to enable the species switching procedures (see §2.5.3.3).

`<integer: 1000>  # dWorkspace_c:  Double workspace for chemistry`
The amount of Fortran double precision workspace to allocate for the chemistry routines. The chemistry routines will abort if this value is too small. In that case, raise its value and rerun the code. The amount of workspace needed depends on the number of species and the reactions, but not on the grid size.

```
<integer: 1000>    # iWorkspace_c:  Integer workspace for chemistry
```
The amount of Fortran integer workspace to allocate for the chemistry routines. The chemistry routines will abort if this value is too small. In that case, raise its value and rerun the code. The amount of workspace needed depends on the number of species and the reactions, but not on the grid size.

5.3 Time Control

```
$S TIME
<real [T]: 0>    # tInitial:  Initial time
```
The initial or starting time.

```
<real [T]>    # tFinal:  Final time
```
The final or ending time.

5.3.1 Flow

```
$SS Flow
# This sectional unit is needed only if computeFlow != 0
<integer>    # nStepsMax_f:  Maximum number of flow steps
```
The maximum number of flow steps allowed.

```
<integer>    # ndtMax_f:  Number of flow time step sizes
```
The maximum number of flow step size increments given below.

```
{
# For each of the ndtMax_f time step sizes, give the following
```
The data are the time it begins and its size. The first time step size will be assumed to be valid at the initial time. The times must increase monotonically.

```
<real [T]>    # tdtMax_f:  Time of onset
```
Time of onset of the next time step size.

```
<real [T]>    # dtMax_f:  Time step size
```
The maximum allowable time step size.

```
}
```

5.3.2 Transport

```
$SS Transport
# This sectional unit is needed only if computeTransport != 0,
# computeChemistry != 0, or computeRND != 0
<integer>    # nStepsMax_t:  Maximum number of transport steps
```
The maximum number of transport steps allowed.

```
<integer>    # nStepsMax_t_per_f:  Maximum number of transport steps per flow step
```
The maximum number of transport steps allowed per flow step. A negative number is intrepreted as infinity.

```
<integer>    # ndtMax_t:  Number of transport time step sizes
```
The maximum number of transport step size increments given below.

```
{
```

`# For each of the ndtMax_t time step sizes, give the following`

The data are the time it begins and its size. The first time step size will be assumed to be valid at the initial time. The times must increase monotonically.

`<real [T]>  # tdtMax_t:  Time of onset`

Time of onset of the next time step size.

`<real [T]>  # dtMax_t:  Time step size`

The maximum allowable time step size.

}

5.4 Spatial Grid

`$S GRID`

`<integer> <integer> <integer>  # nx, ny, nz:  Number of grid cells`

The number of grid cells (i.e., elements) in each coordinate direction.

`<integer> <integer> <integer>  # periodicBC_x, _y, _z:  Boolean for periodicity`

Set to 1 if the grid covers a periodic domain in the given coordinate direction (i.e., if there are periodic boundary conditons).

`0 || 1  # zIsDepth:  Direction of the z coordinate`

Boolean for whether the z coordinate points down (depth) or up (height); that is, whether z increases with depth or with height. We remark that internal to the code z points down.

`uniform || rectangular || xy-rectangular || {  # gridType:`
`  # uniform, rectangular, xy-rectangular, or {`

The type of grid that is to be specified. The grid must be rectangular or logically rectangular. The options are:

- uniform for uniformly rectangular (i.e., each cell has the same dimensions);
- rectangular for nonuniformly rectangular (i.e., each cell is rectangular, but they may vary in size);
- xy-rectangular for a grid that is nonuniform rectangular in x and y but fully variable in z;
- { to specify a fully logically rectangular grid (this is the beginning of a data block).

This must be followed by one of the following sets of input data items.

5.4.1 Uniform Rectangular Grid

`# Use the following when gridType is uniformgrid`

`<real [L]> <real [L]>  # xMin, xMax:  Minimal and maximal x points`

The minimal and maximal x points of the computational grid.

`<real [L]> <real [L]>  # yMin, yMax:  Minimal and maximal y points`

The minimal and maximal y points of the computational grid.

`<real [L]> <real [L]>  # zMin, zMax:  Minimal and maximal z points`

The minimal and maximal z points of the computational grid.

5.4.2 Nonuniform Rectangular Grid

```
# Use the following when gridType is rectangular
{
# For each of the nx+1 x grid points, give the following
<real [L]>    # xp:  The x grid points
}
{
# For each of the ny+1 y grid points, give the following
<real [L]>    # yp:  The y grid points
}
{
# For each of the nz+1 z grid points, give the following
<real [L]>    # zp:  The z grid points
}
```

5.4.3 An XY-rectangular, Z-variable Grid

```
# Use the following when gridType is xy-rectangular
{
# For each of the nx+1 x grid points, give the following
<real [L]>    # xp:  The x grid points
}
{
# For each of the ny+1 y grid points, give the following
<real [L]>    # yp:  The y grid points
}
{
# For each of the (nx+1)*(ny+1)*(nz+1) z grid points, give the following
<real of a data block [L]>    # zp:  The z grid points
```

See §4.7.3 for information on a data block.

```
}
```

5.4.4 A Fully Logically Rectangular Grid

```
# Use the following when gridType is {
```

This indicates the beginning of a data block

```
# For each of the (nx+1)*(ny+1)*(nz+1) x grid points, give:
<real of a data block [L]>    # xp:  The x grid points
```

See §4.7.3 for information on a data block.

}

{
```
# For each of the (nx+1)*(ny+1)*(nz+1) y grid points, give the following
<real of a data block [L]>    # yp:  The y grid points
```
See §4.7.3 for information on a data block.

}

{
```
# For each of the (nx+1)*(ny+1)*(nz+1) z grid points, give the following
<real of a data block [L]>    # zp:  The z grid points
```
See §4.7.3 for information on a data block.

}

5.5 Porous Medium Material Properties

```
$S MATERIAL PROPERTIES
<real [L/T^2]:  9.8 [m/sec^2]>    # gravity:  Gravitational constant
```
The gravitational constant.

```
<real grid array [1]>    # porosity:  Porosity
```
A grid array of porosity values, given over the 3-D domain as cell-centered data, one per cell (see §4.7.3). An immobile phase can be accounted for by decreasing the porosity of the rock itself and entering those values here instead. Herein, pore volume refers to flowing phase volume.

5.5.1 Permeabilities (or Conductivities)

```
$SS Permeability || $SS Conductivity
# This sectional unit is needed only if computeFlow != 0
```
The meaning of the input "permeability" (`perm`) values is either the absolute permeability or the hydraulic conductivity, depending on the subsection name. The hydraulic conductivity is $\mathbf{K} = \rho g \mathbf{k}/\mu_0$. Permeability is assumed internal to the code. If an immobile fluid phase is present, its effect on the flowing phase should be reflected in the values given below for the permeability `perm` (i.e., give the product of the absolute permeability and the endpoint relative permeability, converted to conductivity if necessary).

```
scalar || diagonal || symmetric || face    # permType:
        # scalar, diagonal, symmetric, or face
```
Declare the "permeability" to be a scalar (one value per cell), diagonal tensor (three values per cell), a full symmetric tensor (six values per cell), or face centered diagonal tensor values (a single value for each face of the grid). This must be followed by one of the following sets of input data items.

5.5.1.1 Scalar Permeabilities

```
# Use the following when permType is scalar
<real grid array [L^2 (permeability) or L/T (conductivity)]>    # perm:
        # Scalar permeabilities
```

A grid array of scalar permeability values, given over the 3-D domain as cell-centered data, one per cell (see §4.7.3).

5.5.1.2 Diagonal Tensor Permeabilities

```
# Use the following when permType is diagonal
by cells || by components    # permGrouping:
        # Group by grid cells or by tensor components
```

Give the entire permeability tensor for each grid cell, or give successively a single component of the tensor for the entire grid. The word is optional.

```
xx yy zz || <any permutation>    # permComponentOrder:
        # Order of tensor's components
```

Declare the order of the permeability tensor component data.

```
<1 or 3 real grid arrays [L^2 (permeability) or L/T (conductivity)]>    # perm:
        # Diagonal tensor permeabilities
```

One or three grid arrays of diagonal permeability values, given over the 3-D domain as cell-centered data, three or one number per cell (see §4.7.3). If the `permGrouping` is cells, give the three components of the tensor (as ordered by `permComponentOrder`) for each grid cell. Otherwise, give three grid arrays of single item cell-centered data, one for each tensor component (again, as ordered by `permComponentOrder`).

5.5.1.3 Symmetric Tensor Permeabilities

```
# Use the following when permType is symmetric
cells || components    # permGrouping:
        # Group by cells or by tensor components
```

Give the entire permeability tensor for each grid cell, or give successively a single component of the tensor for the entire grid.

```
xx yy zz xy xz yz || <any permutation>    # permComponentOrder:
        # Order of tensor's components
```

Declare the order of the permeability tensor component data.

```
<1 or 6 real grid arrays [L^2 (permeability) or L/T (conductivity)]>    # perm:
        # Symmetric tensor permeabilities
```

One or six grid arrays of symmetric tensor permeability values, given over the 3-D domain as cell-centered data, six or one number per cell (see §4.7.3). If the `permGrouping` is cells, give the six components of the tensor (as ordered by `permComponentOrder`) for each grid cell. Otherwise, give six grid arrays of single item cell-centered data, one for each tensor component (again, as ordered by `permComponentOrder`).

5.5.1.4 Face Permeabilities

```
# Use the following when permType is face
<3 real grid arrays [L^2 (permeability) or L/T (conductivity)]>    # perm:
        # Face permeabilities
```

Three grid arrays of permeability values, given over the 3-D domain as face-centered data, one per face (see §4.7.3). The arrays give values for the x-faces, y-faces, and z-faces and are of size `(nx+1,ny,nz)`, `(nx,ny+1,nz)`, and `(nx,ny,nz+1)`), respectively.

5.5.2 Dispersion

```
$SS Dispersion
# This sectional unit is needed only if computeTransport != 0
```

The diffusion/dispersion tensor is defined above in §2.3.1 on page 8.

```
<integer>    # uniformDispersion:  Flag for uniform dispersion
```

Flag for uniform dispersion if true (i.e., 1) or nonuniform if false (i.e., 0).

```
# Use this syntax if uniformDispersion is true (i.e., 1)
<real [L^2/T]>    # molDiff:  Molecular diffusion
```

The molecular diffusion coefficient.

```
<real [L]>    # longDisp:  Longitudinal dispersion
```

The longitudinal dispersion coefficient.

```
<real [L]>    # transDisp:  Transverse dispersion
```

The transverse dispersion coefficient.

```
# Use this syntax if uniformDispersion is false (i.e., 0)
<real grid array [L^2/T]>    # molDiffAry:  Molecular diffusion array
```

The molecular diffusion coefficient array.

```
<real grid array [L]>    # longDispAry:  Longitudinal dispersion array
```

The longitudinal dispersion coefficient array.

```
<real grid array [L]>    # transDispAry:  Transverse dispersion array
```

The transverse dispersion coefficient array.

5.5.3 Linear Sorption/Retardation

```
$SS Sorption
# This sectional unit is needed only if computeTransport != 0,
# computeChemistry != 0, or computeRND != 0
```

The linear sorption model is defined above in §2.3 and §2.3.2 (see pages 7 and 8).

```
<integer>    # nSorpTypes:  Number of sorption types
```

The number of distinct sorption (or retardation) capacities.

```
{
# For each of the nSorpTypes sorption types, give the following
```

This section induces an ordering of the sorption capacities that is used elsewhere.

`<real grid array [1]> || porosity    # sorp:  Sorption capacities`

The sorption capacities (σ). The option "`porosity`" implies that the retardation is proportional to `porosity`. Otherwise, give one value per cell (see §4.7.3).

}

5.6 Phase Properties

`$S PHASE PROPERTIES`

`<real [M/(LT)]>    # fluidViscosity:  Flowing phase viscosity`

The flowing phase viscosity of the resident fluid.

`<real [M/L^3]>    # fluidDensity:  Flowing phase mass density`

The (average) flowing phase mass density. The fluid is assumed incompressible, so the density of the fluid is not affected by its composition (i.e., we assume dilute solutions). This can cause inconsistencies if heavy or light components are present in the fluid in high concentrations.

`<integer>    # nPhases:  Number of phases`

The total number of fluid and solid phases.

5.6.1 Phase Description

`$SS Phase`
`# Include one such sectional unit per phase as specified by nPhases`

This section induces an ordering of the phases that is used elsewhere.

`<word of characters>    # phaseNames:  Phase name`

The name of the phase. Maximum of 15 characters.

`multi-species || single-species    # phaseType:  multi-species or single-species`

Whether the phase consists of multiple chemical species or a single chemical species. A multi-species phase is typically the flowing phase; single-species phases are typically minerals.

`<real [1/L^3]> || <omit>    # chmPhaseDensity:  Molar density`

The molar density of the phase, in moles per fluid phase volume. This item occurs only if `phaseType` is multi-species.

5.7 General Chemistry Properties

`$S CHEMISTRY`
`# This sectional unit is needed only if computeChemistry > 0`

`<real [temperature reading]>    # absTemp:  Equilibrium temperature`

The temperature at which equilibrium is computed. See §4.7.1 for temperature readings. This value is currently not used.

`<real [M/(LT^2]>    # chemPres:  Equilibrium pressure`

A representative pressure at which equilibrium is computed. This value is currently not used.

`ideal || non-ideal    # ideal:  ideal or non-ideal`

Indicate whether all phases of the solution are to be considered ideal or non-ideal. Currently, only the ideal option is implemented.

5.8 Chemical Species Properties

`$S CHEMICAL SPECIES`

`<integer>  # nComps:  Number of components`

The number of chemical species that are components of the general chemistry reactions.

`<integer>  # nProducts:  Number of products`

The number of chemical species that are products of the general chemistry reactions.

`# nSpecies = nComps + nProducts`

For ease of exposition and internal to the code, we let `nSpecies` be the sum of `nComps` and `nProducts`. If `computeChemistry` ≤ 0, so general chemistry is not used, `nSpecies` is the number that is important, and it can be given by dividing arbitrarily the species into components and products above.

5.8.1 Species Description

`$SS Component || $SS H Component || $SS H2O Component || $SS Product`
`# Include one such sectional unit per component as specified by nComps`
`# Follow by one such sectional unit per product as specified by nProducts`

General chemistry requires knowledge of whether the species is a fundamental chemical component or a product of a reaction, and sometimes requires knowledge of where the hydrogen or water component is.

This sectional unit induces an ordering of the components, products, and species that is used elsewhere. Note that the species order places all components before all products; that is, the component order index is the same as the species order index, and the product order index plus `nComps` is the species order index. Species are numbered beginning with 1.

`<word of characters>  # specieName:  Specie name`

The name of the specie. Maximum of 15 characters.

`<real [M]>  # molecularWeight:  Molecular weight`

The molecular weight of the specie, in mass units per mole. This is used only for computing radionuclide decay and miscible displacement.

`<real [1]>  # phaseDist:  The alpha parameter in Henry's Law`

The Henry's Law constant for linear sorption. The linear sorption model is defined above in §2.3 and §2.3.2 (see pages 7 and 8). This is the parameter α_i. A value of zero indicates that no sorption takes place. A negative value indicates an immobile specie, and the value of $\alpha_i\sigma_i$ is assumed to be 1.

`<integer> || <omit>  # sorpType:  Sorption type`

If needed (`phaseDist` > 0), the sorption capacity number from 1 to `nSorpTypes` specified earlier. This uses the order of the sorption capacities induced above.

`$SSS Chemistry`
`# This sectional unit is needed only if computeChemistry > 0`

Components and products are described differently. The command line argument `-c` can be used to see how a given data set is interpreted.

5.8.1.1 Component Chemistry Specification

```
# Use this syntax if the subsection heading is Component,
# H Component, or H2O Component
<integer>    # phaseIdentity:  Participating phase
```

The phase in which the component participates, based on the organization of phases given in elsewhere.

```
<real [1]>    # compCharge:  Charge
```

The intrinsic charge of the component. Used to compute the intrinsic species charges, and it can be valuable for debugging the stoichiometry.

5.8.1.2 Product Chemistry Specification

```
# Use this syntax if the subsection heading is Product
<integer>    # phaseIdentity:  Participating phase
```

The phase in which the product participates, based on the organization of phases given in elsewhere.

```
{
# For each of the nComps components, give the following
<real [1]>    # stoich:  Stoichiometry formula number
```

The formula number for this product species for the next component.

```
}
0 || 1 || 2    # reactionType:  Reaction type
```

Flag to select the class of reaction. The command line argument `-t` gives the options:

0 for equilibrium controlled;
1 for mass-action type kinetic;
2 for Monod type kinetic.

Equilibrium Reaction

```
# Use this syntax if reactionType specifies an equilibrium reaction
<real [1]>    # pK: Log 10 equilibrium constant
```

The base 10 logarithm of the equilibrium constant.

Mass-action Type Kinetic Reaction

```
# Use this syntax if reactionType specifies a mass-action type kinetic reaction
<real [1]> <real [1]>    # pKf, pKb:  Log 10 forward and backward rate-constants
```

For Kinetic reactions, give the base 10 logarithm of the forward and backward rate-constants.

```
{
# For each of the nComps components, give the following
<real [1]>    # stoichK: Rate-law powers
```

Powers on the component-concentration in the rate-law. If taken equal to `stoich`, the "classical" form of the rate-law results.

```
}
```

Monod Type Kinetic Reaction

```
# Use this syntax if reactionType specifies a monod type kinetic reaction
<real [1]> <real [1]: -99>    # pKf, pKb:  Log 10 forward & backward rate-constants
```

For Kinetic reactions, give the base 10 logarithm of the forward and backward rate-constants. The backward rate-constant is included for generality; it is often not used in biochemical pathways, and a small value (e.g., -99) should be assigned.

```
{
# For each of the nComps components, give the following
<real [1]>    # stoichK: Rate-law powers
```

Powers on the component-concentration in the rate-law (2.16) (§2.5.2.2) in the form c_j^p. Components which appear in an expression of the form $\frac{c_j}{K+c_j}$ are treated separately (see next set of inputs) and should be given a power of 0 here.

```
}
```

```
{
# For each of the nComps components, give the following
<real [1]>    # halfSatConst:  Half saturation constants
```

The so-called "half-saturation constants" in Monod kinetics (2.16) (§2.5.2.2) which are the constants K in terms of the form $\frac{c_j}{K+c_j}$. A component that does not appear in the rate expression in this form should be given a value of 0 here.

```
}
```

5.9 Miscible Displacement

```
$S MISCIBLE DISPLACEMENT
# This sectional unit is needed only if computeFlow != 0 and
# computeTransport != 0, computeChemistry != 0, or computeRND != 0
0 || 1    # modelMiscDisp:  Miscible displacement model
```

Flag declaring how the fluid viscosity will be computed in the flow computation. The command line argument `-t` gives the options:

0 for a the fixed viscosity `fluidViscosity`;
1 for the quarter power mixing rule.

5.9.1 Quarter Power Mixing Rule

```
# Use the following when modelMiscDisp selects
# the quarter power mixing rulemodelMiscDisp
```

This rule is defined above in §2.2.1 on page 7.

```
{
# For each of the nSpecies species, give the following
```

`<real [1]>  # viscosityRatio:  Viscosity ratio`

The ratio of the viscosity of the species to the resident fluid viscosity `fluidViscosity` μ/μ_0. Use a zero viscosity ratio to indicate a species that contributes to the mass fraction computations, but not to the viscosity directly (i.e., the ratio is infinity). Ignore completely any species with a negative viscosity ratio.

}

5.10 Radionuclide Decay

`$S RADIONUCLIDE DECAY`
`# This sectional unit is needed only if computeRND != 0`

`<integer>  # nChains:  Number of RND chains`

The number of distinct radionuclide decay chains.

`$SS Chain`
`# Include one such sectional unit per chain as specified by nChains`

This sectional unit uses the ordering of the species that is defined elsewhere.

`<integer>  # chainLen:  Length of chain`

The number of species in the chain.

`linear || branching  # chainType:  linear or branching`

State whether the chain is a simple linear chain (each species decays into only one other species) or a branching chain (each species can decompose into all succeeding species).

{
`# For each of the chainLen species, give the following`

`<integer>  # specieNumber:  Specie number`

Give the next species in the chain. This makes use of the order of the species given elsewhere.

`<real [T]>  # halfLife:  Half-life`

The half-life (stable is indicated by a negative half-life). The half-lives of the species in the chain *must* be unique.

`<omit> || <real [1]> ...  # branchRatios:  Branch ratios`

If `chainType` is `branching`, give the branching ratio of the first, second, third, ... member of the chain into this specie (there are none for the first, one for the second, etc.).

}

5.11 Specialized Reactions

`$S SPECIALIZED REACTIONS`
`# This sectional unit is needed only if computeChemistry < 0`

If `computeChemistry` ≥ 0, this section will be skipped. The specialized reaction module is intended to be modifiable by the user, so this section is quite generic. The user provides nonequilibrium and nonequilibrium reaction functions in the C-code file rxn.c or in the Fortran code file rxnf.f. A single C preprocessor directive in rxn.c must be set to select whether the reaction functions are written in C or Fortran. This sectional unit uses the ordering of the species that is defined elsewhere.

`<integer>  # rxn_nComps: Number of component species in reactions`

The number of species that take part in the reactions. This number may be less than `nSpecies`.

{
`# For each of the rxn_nComps component species, give the following`

`<integer>  # rxn_comp: Species number of the next reaction component`

The species involved in the reactions are selected here, and possibly reordered. Order according to how the reaction functions in rxn.c or rxnf.f expect them. Species are numbered beginning with 1, and ordered as defined elsewhere.

}

`<integer>  # rxn_nMicroSteps: Number of reaction micro steps`

The number of sub steps to take in solving the reactions per transport time step.

`<double [1]>  # rxn_tolerance: Convergence tolerance parameter`

The Runge-Kutta-Fehlberg convergence tolerance parameter used to control the estimated error.

`<integer>  # rxn_nDParams: Number of real parameters for user's rxn fcns`

The number of real parameters needed to describe the specialized reactions.

{
`# For each of the rxn_nDParams real parameters, give the following`

`<real [1]>  # rxn_dParams: Real parameter`

The real (double) reaction parameters. These are passed to the reaction functions in rxn.c or rxnf.f.

}

`<integer>  # rxn_nIParams: Number of integer parameters for user's rxn fcns`

The number of integer parameters needed to describe the specialized reactions.

{
`# For each of the rxn_nIParams integer parameters, give the following`

`<integer>  # rxn_iParams: Integer parameter`

The integer reaction parameters. These are passed to the reaction functions in rxn.c or rxnf.f.

}

5.12 Initial Conditions

`$S INITIAL CONDITIONS`

5.12.1 Flow

`$SS Flow`
`# This sectional unit is needed only if computeFlow = 0`

`<real grid array [L/T]>  # velX: X velocity`

A grid array of x-Darcy velocity values, given over the 3-D domain as point-centered data in x and cell-centered data in y and z, for a total of $(\texttt{nx}+1)*\texttt{ny}*\texttt{nz}$ velocities (see §4.7.3).

`<real grid array [L/T]>  # velY: Y velocity`

A grid array of y-Darcy velocity values, given over the 3-D domain as point-centered data in y and cell-centered data in x and z, for a total of $\texttt{nx}*(\texttt{ny}+1)*\texttt{nz}$ velocities (see §4.7.3).

```
<real grid array [L/T]>    # velZ: Z velocity
```

A grid array of z-Darcy velocity values, given over the 3-D domain as point-centered data in z and cell-centered data in x and y, for a total of $\texttt{nx} * \texttt{ny} * (\texttt{nz} + 1)$ velocities (see §4.7.3).

5.12.2 Transport

```
$SS Transport
# This sectional unit is needed only if computeTransport != 0,
# computeChemistry != 0, or computeRND != 0
{
# For each of the nSpecies species, give the following
<real grid array [1/L^3]>    # conc:  Molar concentration
```

A grid array of initial molar concentration values, given over the 3-D domain as cell-centered data, one per cell (see §4.7.3). The units specification facilities (see §4.7.1) can be used to convert mass (or other) concentrations to molar concentrations.

```
}
```

5.13 Boundary Conditions

```
$S BOUNDARY CONDITIONS
<integer>    # nBCRegions:  Number of boundary regions
```

The number of distinct regions of the domain boundary.

```
<integer>    # maxnBCInterp_f:  Maximum number of flow interpolation times
```

The maximum number of interpolation times used to specify the flow boundary conditions (see below).

```
<integer>    # maxnBCInterp_t:  Maximum number of transport interpolation times
```

The maximum number of interpolation times used to specify the transport boundary conditions (see below).

The following grid arrays define an order to the boundary regions used elsewhere.

```
# The following lines appear only if periodicBC_x is 0
<integer grid array>    # bcRegion_x_min:  Boundary region map
```

A grid array of boundary region identification numbers from 1 to `nBCRegions`, given over the 2-D boundary face for minimal x. This is cell-centered data, one per cell face (see §4.7.3), for a total of $\texttt{ny} * \texttt{nz}$ integers.

```
<integer grid array>    # bcRegion_x_max:  Boundary region map
```

A grid array of boundary region identification numbers from 1 to `nBCRegions`, given over the 2-D boundary face for maximal x. This is cell-centered data, one per cell face (see §4.7.3), for a total of $\texttt{ny} * \texttt{nz}$ integers.

```
# The following lines appear only if periodicBC_y is 0
<integer grid array>    # bcRegion_y_min:  Boundary region map
```

A grid array of boundary region identification numbers from 1 to `nBCRegions`, given over the 2-D boundary face for minimal y. This is cell-centered data, one per cell face (see §4.7.3), for a total of $\texttt{nx} * \texttt{nz}$ integers.

```
<integer grid array>    # bcRegion_y_max:  Boundary region map
```
A grid array of boundary region identification numbers from 1 to `nBCRegions`, given over the 2-D boundary face for maximal y. This is cell-centered data, one per cell face (see §4.7.3), for a total of `nx` $*$ `nz` integers.

```
# The following lines appear only if periodicBC_z is 0
<integer grid array>    # bcRegion_z_min:  Boundary region map
```
A grid array of boundary region identification numbers from 1 to `nBCRegions`, given over the 2-D boundary face for minimal z. This is cell-centered data, one per cell face (see §4.7.3), for a total of `nx` $*$ `ny` integers.

```
<integer grid array>    # bcRegion_z_max:  Boundary region map
```
A grid array of boundary region identification numbers from 1 to `nBCRegions`, given over the 2-D boundary face for maximal z. This is cell-centered data, one per cell face (see §4.7.3), for a total of `nx` $*$ `ny` integers.

5.13.1 Boundary Region Specification

```
$SS Region
# Include one such sectional unit per boundary region as specified by nBCRegions
```

These regions are mapped to the physical boundary and ordered by the `bcRegion` grid arrays.

5.13.1.1 Flow

```
$SSS Flow
# This sectional unit is needed only if computeFlow != 0
0 || 1 || 2    # bcType_f:  BC type
```
Flag declaring the type of the boundary condition for flow. The command line argument `-t` gives the options:

0 for no flow condition, $\mathbf{u} \cdot \nu = 0$ (0 normal velocity);
1 for a Neumann condition $-\mathbf{u} \cdot \nu = q_{\mathrm{N}}$ (specified normal velocity, where the inflow condition is positive and the outflow condition is negative);
2 for a Dirichlet condition $p = p_{\mathrm{N}}$ (specified pressure).

Above, q_{N} or p_{N} needs to be specified as a function of time.

Function of Time Specification

```
# The following lines appear only if a function of time
# needs to be specified; that is, bcType_f != 0
<omit> || pressure || potential || head || hydraulicHead    # bcPresType_in:
        # Dirichlet pressure, potential, head, or hydraulicHead
```
If `bcType_f` selects the Dirichlet condition, give the meaning of the variable to which the condition applies (see page 49 for a discussion on the meaning of the options).

```
<integer>    # nBCInterp_f:  Number of interpolation times
```
The number of interpolation times used in the definition of the boundary condition as a piecewise continuous linear function of time. Constant extrapolation is used before and after the first and last times; thus, a single interpolation time gives a constant condition. The times must not decrease; however, two pairs with the same time represent a discontinuous jump.

{
# For each of the nBCInterp_f interpolation times, give the following

<real [T]> # bc_f_time: BC time

The time at which the next interpolation value is to be used.

<real [L/T (Neumann) or M/(L*T**2) (Dir pres/pot) or L (Dir head/hHead)]>
  # bc_f_value: BC value

The next interpolation value.

}

5.13.1.2 Transport

$SSS Transport
This sectional unit is needed only if computeTransport != 0

0 || 1 || 2 # bcType_t: BC type

Flag declaring the type of the boundary condition for transport. The command line argument `-t` gives the options:

0 for no flow condition or outflow condition, where it should be the case that $\mathbf{u} \cdot \nu \geq 0$ and then $\mathbf{D}\nabla c \cdot \nu = 0$ (0 normal dispersive flux);
1 for an inflow condition (i.e., a Robin condition), where it should be the case that $\mathbf{u} \cdot \nu \leq 0$ and then $(\mathbf{u}c - \mathbf{D}\nabla c) \cdot \nu = c_{\mathrm{B}}\mathbf{u} \cdot \nu$ (specified normal advective flux);
2 for a Dirichlet condition $c = c_{\mathrm{B}}$ (specified concentration).

Above, c_{B} needs to be specified as a function of time.

Function of Time Specification

The following lines appear only if a function of time
needs to be specified; that is, bcType_t != 0

{
# For each of the nSpecies species, give the following

<integer> # nBCInterp_t: Number of interpolation times

The number of interpolation times used in the definition of the boundary condition as a piecewise continuous linear function of time. Constant extrapolation is used before and after the first and last times; thus, a single interpolation time gives a constant condition. The times must not decrease; however, two pairs with the same time represent a discontinuous jump.

{
# For each of the nBCInterp_t interpolation times, give the following

<real [T]> # bc_t_time: BC time

The time at which the next interpolation value is to be used.

<real [1/L^3]> # bc_t_value: BC conc value

The next interpolation value.

}

}

5.14 Well Specification

```
$S WELLS
<integer>    # nWells:  Number of wells
```
The number of wells. Note that if there are wells, `nDom_z` *must* be set to 1 above.

```
<integer>    # maxnWellInterp_f:  Maximum number of flow interpolation times
```
The maximum number of interpolation times used to specify the flow well conditions (see below).

```
<integer>    # maxnWellInterp_t:  Maximum number of transport interpolation times
```
The maximum number of interpolation times used to specify the transport well conditions (see below).

5.14.1 Single Well Description

```
$SS Well
# Include one such sectional unit per well as specified by nWells
0 || 1 || 2 || 3 || 4    # wellType:  Well type
```
Flag specifying the type of the well. The command line argument `-t` gives the options:

0 for an inactive well;
1 for an injection well;
2 for an extraction well (or production well);
3 for an bottomhole production well (that uses the Peaceman correction [23]);
4 for an reinjection well (or sparging well).

A reinjection well has both an extraction and an injection interval; the fluid extracted is reinjected elsewhere. An inactive well is equivalent to no well at all, and no further specification should be given below.

```
# If the well is not inactive, include the following information
<real [L]>    # wellRadius:  Well radius
```
The radius of the well.

```
<integer> <integer>    # hWellIndex:  The x and y indices of well
```
The location of the well in the grid, given as the x and y cell-centered data index numbers. The well is generally assumed to occupy entire cells.

```
<integer>    # zWellIndexLo:  Lower z index of well
```
The top (if `zIsDepth` is true) or bottom (otherwise) of the well in the grid, given as the lower z cell-centered data index number. If `wellType` selects a reinjection well, specify the injection part.

```
<integer>    # zWellIndexHi:  Higher z index of well
```
The bottom (if `zIsDepth` is true) or top (otherwise) of the well in the grid, given as the upper z cell-centered data index number. If `wellType` selects a reinjection well, specify the injection part.

```
<omit> || <integer>    # zWellIndexLo:  Lower z index of well
```
If `wellType` selects a reinjection well, specify the lower z index number of the extraction part.

```
<omit> || <integer>    # zWellIndexHi:  Higher z index of well
```
If `wellType` selects a reinjection well, specify the upper z index number of the extraction part.

5.14.1.1 Flow

```
$SSS Flow
# This sectional unit is needed only if computeFlow != 0
```

Bottomfole Producer

```
# Use if wellType selects a bottomhole producer
<real [M/(L*T^2)]>    # wellPres:  Well pressure potential
```

The pressure potential of the well (see §2.1 on page 6 for the meaning of potential).

Other Active Well Types

```
# The following lines appear only if wellType does not select a bottomhole producer

<integer>    # nWellInterp_f:  Number of interpolation times
```

The number of interpolation times used in the definition of the wells as a piecewise continuous linear function of time. Constant extrapolation is used before and after the first and last times; thus, a single interpolation time gives a constant condition. The times must not decrease; however, two pairs with the same time represent a discontinuous jump.

```
{
# For each of the nWellInterp_f interpolation times, give the following
<real [T]>    # wellInj_f_time:  Well time
```

The time at which the next interpolation value is to be used.

```
<real [L^3/T]>    # wellInj_f_value:  Well injection or extraction rate
```

The next interpolation value, volume of injected or extracted fluid per unit time. It should be nonnegative.

```
}
```

5.14.1.2 Transport

```
$SSS Transport
# This sectional unit is needed only if computeTransport != 0 and
# wellType does not select an extractor or a bottomhole producer

{
# For each of the nSpecies species, give the following
<integer>    # nWellInterp_t:  Number of interpolation times
```

The number of interpolation times used in the definition of the boundary condition as a piecewise continuous linear function of time. Constant extrapolation is used before and after the first and last times; thus, a single interpolation time gives a constant condition. The times must not decrease; however, two pairs with the same time represent a discontinuous jump.

```
{
# For each of the nWellInterp_t interpolation times, give the following
<real [T]>    # wellInj_t_time:  Well time
```

The time at which the next interpolation value is to be used.

```
<real [1/L^3]>    # wellInj_t_value:  Well conc value
```
The next interpolation value, the molar concentration of the injected fluid.

}

}

5.15 Leaking Source Specification

```
$S LEAKS
<integer>    # nLeaks:  Number of leaks
```
The number of leaking sources.

```
<integer>    # maxnLeakInterp:  Maximum number of interpolation times
```
The maximum number of interpolation times used to specify the leak conditions (see below).

5.15.1 Single Leak Description

```
$SS Leak
# Include one such sectional unit per leak as specified by nLeaks
<integer> <integer> <integer>    # leakIndexLo:  Lower cell of leak
```
The leak occurs over a rectangular region. Specify the three indices of the lower cell of the leak in the grid, given as cell-centered data indices.

```
<integer> <integer> <integer>    # leakIndexHi:  Upper cell of leak
```
The leak occurs over a rectangular region. Specify the three indices of the upper cell of the leak in the grid, given as cell-centered data indices.

{
```
# For each of the nSpecies species, give the following
<integer>    # nLeakInterp:  Number of interpolation times
```
The number of interpolation times used in the definition of the leak as a piecewise continuous linear function of time. Constant extrapolation is used before and after the first and last times; thus, a single interpolation time gives a constant condition. The times must not decrease; however, two pairs with the same time represent a discontinuous jump.

{
```
# For each of the nLeakInterp interpolation times, give the following
<real [T]>    # leakInj_t_time:  Leak time
```
The time at which the next interpolation value is to be used.

```
<real [1/T]>    # leakInj_t_value:  Leak conc value
```
The next interpolation value, the molar rate of fluid leaking into the domain.

}

}

5.16 Output

`$S OUTPUT`

`<integer>  # runNumber:  Run number`

The number of the current run. (Currently not used.)

`<line of characters describing the run>`
`# runDescription:  Title or description`

A title or description of the current run. Maximum of 120 characters, including spaces. (Currently not used.)

`0 || 1  # verbosity:  Driver verbosity flag`

Boolean to monitor progress in the driver routines. A minimal amount of monitoring output will always be given.

`0 || 1 || 2 || 3 || 4  # debug:  Driver debug flag`

Flag to set the debug mode in the driver routines. The command line argument `-t` gives the options:

0 for no debugging tests or output;
1 for debugging tests of the parallel set-up;
2 for debugging output of miscellaneous items;
3 for debugging output (dump) of the input data variables;
4 for debugging output (dump) of the internal global variables.

The debugging level chosen includes everything at a lower level. Level 2 (and above) produces output. Output is written by each processor into its own file, called `debug????`, where the processor number replaces the question marks.

`<word of characters>  # outDir:  Output directory`

The directory in which to write output data files. Use the `$LITERAL` command if the name begins with a special symbol. The current directory is '`.`' in unix. Maximum of 120 characters.

`0 || 1  # outFormat3D: Output 3-D file format`

Flag selecting the format for 3-D output data files. The command line argument `-t` gives the options:

-1 Eye visualization package format;
0 raw data format;
1 Tecplot visualization package format.

`<integer>  # nSpeciesPerOutfile:  Number of species per 3-D output file`

If multiple species can appear in a single 3-D output file, this sets the maximum number allowed.

`0 || 1  # initialOut:  Output 3-D initial conditions flag`

Boolean for giving output data files at the initial time.

`1  # outFormat1D: Output 1-D file format`

Flag selecting the format for 1-D output data files. The command line argument `-t` gives the options:

1 Tecplot visualization package format.

5.16.1 Flow

```
$SS Flow
# This sectional unit is needed only if computeFlow != 0
<integer>    # dStepOut_pres:  Steps between pressure outputs
```

The number of flow steps between writing pressure output data files. The value -1 is interpreted as positive infinity (i.e., generate no output due reaching a certain step number).

```
<integer>    # dStepOut_vel:  Steps between velocity outputs
```

The number of flow steps between writing velocity output data files. The value -1 is interpreted as positive infinity (i.e., generate no output due reaching a certain step number).

```
<real [T]>    # dtOut_pres:  Time between pressure outputs
```

The flow time increment between writing pressure output data files. The value -1 is interpreted as positive infinity (i.e., generate no output due reaching a certain time).

```
<real [T]>    # dtOut_vel:  Time between velocity outputs
```

The flow time increment between writing velocity output data files. The value -1 is interpreted as positive infinity (i.e., generate no output due reaching a certain time).

```
potential || pressure || hydraulicHead || head    # presType_out:
        # potential, pressure, hydraulicHead, or head
```

The physical interpretation of the "`pressure`" variable. The meaning is as follows:

- `potential` for the flow potential $\psi = p - \rho g z$ [M/(LT^2)];
- `pressure` for p [M/(LT^2)];
- `hydraulicHead` for $H = \psi/(\rho g)$ [L];
- `head` for the head $h = p/(\rho g)$ [L].

Internal to the code, the variable `pressure` is assumed to be "potential."

```
<integer 0-1000>    # outFlags_f_1:  Level of flow verbosity
```

Flag to set the level of computation monitoring in the flow routines. Level 5 gives a summary of the iterations used to solve the interface problem of the domain decomposition solver.

```
<integer 0-1000>    # outFlags_f_2:  Level of flow debugging
```

Flag to set the level of debugging output in the flow routines.

5.16.2 Transport

```
$SS Transport
# This sectional unit is needed only if computeTransport != 0,
# computeChemistry != 0, or computeRND != 0
<integer>    # dStepOut_conc:  Steps between concentration outputs
```

The number of transport steps between writing concentration output data files. The value -1 is interpreted as positive infinity (i.e., generate no output due reaching a certain step number).

```
<real [T]>    # dtOut_conc:  Time between concentration outputs
```

The transport time increment between writing concentration output data files. The value -1 is interpreted as positive infinity (i.e., generate no output due reaching a certain time).

```
<integer>    # nHistories_t:  Number of 1-D time histories
```

The number of 1-D time output data!concentration history files to write, as described below.

```
0 || 1    # outFlags_t_1:  Monitor transport progress
```

Boolean to select monitoring progress in the transport routines.

`0 || 1    # outFlags_t_2:  Monitor transport time step cuts`
Boolean to select monitoring time step cuts in the transport routines.

`0 || 1    # outFlags_t_3:  Monitor transport dispersion solver`
Boolean to select monitoring iterations of the transport diffusion/dispersion linear solver.

`<integer>    # outFlags_t_4:  Debug transport CMM trace-back points`
Flag to select debugging of the transport trace-back points in the CMM advection scheme (if selected by `computeTransport`). The value 0 means no output, a negative value gives a 3-D plot of points for the Eye visualization package, and a positive integer k between 1 and `nz` gives a logically horizontal 2-D cross-section at index level k for the Draw program.

`0 || 1    # outFlags_t_5:  Debug (write) transport conc`
Boolean to select debugging output for transport `conc` values.

`0 || 1    # outFlags_t_6:  Debug transport call and set-up`
Boolean to select debugging output for transport call and set-up information.

`0 || 1    # outFlags_t_7:  Debug transport CMM trace-back integrals.`
Boolean to select debugging output for the CMM transport trace-back integrals (if the CMM advection scheme is selected by `computeTransport`).

`<integer 0-4>    # outFlags_t_8:  Debug chemistry`
Debug transport reactions. Flag to select debugging output for the chemistry routines. Value 0 gives no information, and increasing amounts of information are given for larger values.

`0 || 1    # outFlags_t_9:  Debug transport dispersion linear system assembly`
Boolean to select debugging output for transport diffusion/dispersion linear system assembly.

5.16.2.1 Concentration Histories

`$SSS History`
`# Include one such sectional unit per history as specified by nHistories_t`

`<integer>    # histType_t:  History type (1=CP,2=CT,3=MP,4=MT)`
The type of output desired. The options are:

1 for the pore volume average of primary phase concentration (CP)
2 for total primary phase moles (CT)
3 for the pore volume average of total moles (MP)
4 for the total moles (MT)

`<integer> <integer> <integer>    # histIndexLo_t_x, _y, _z:  Lower cell of region`
The history sum or average occurs over a rectangular region. Specify the three indices of the lower cell of the region in the grid, given as cell-centered data indices.

`<integer> <integer> <integer>    # histIndexHi_t_x, _y, _z:  Upper cell of region`
The history sum or average occurs over a rectangular region. Specify the three indices of the upper cell of the region in the grid, given as cell-centered data indices.

`<integer>    # nHistSpecies_t:  Number of species in the history`
The number of species to include in the concentration history.

{
`# For each of the nHistSpecies_t species, give the following`

`<integer>    # histSpecie_t:  Species in the history`

}

`<integer>  # dStepOut_hist_t:  Steps between history outputs`

The number of transport steps between writing concentration history output data files. The value -1 is interpreted as positive infinity (i.e., generate no output due reaching a certain step number).

`<real [T]>  # dtOut_hist_t:  Time between history outputs`

The transport time increment between writing concentration history output data files. The value -1 is interpreted as positive infinity (i.e., generate no output due reaching a certain time).

Chapter 6

Input Error Messages

Errors in the input data file(s) can occur for a number of reasons, such as

- the sectional units are out of order;
- key-words or sectional unit names are mispelled;
- the wrong type of data or incorrect physical dimensions are assigned;
- an incorrect number of items appears in a delimited list;
- data items are assigned out of order;
- general syntax is incorrect.

It is easy to understand how the code parses the input file by using the command line argument `-e` or `-E`. This will automatically follow the included files, and it can be extremely helpful in tracking down errors in the input data file(s).

When searching for errors, keep in mind that the error could occur well before the error manifests itself to the code, since it is possible for data, if it is of the proper type, to be assigned to the wrong variables for some time. The actual assignment to variables can be determined by setting the `debug` flag to a high enough value.

There are a series of error messages that may be encountered pertaining to input. Specific error messages will not be described here, but only the following relatively generic error messages.

`Arithmetic operation error in evaluating expression`
: A units expression has an arithmetic error of some kind.

`Beginning of list symbol { not read`
: A user declared number of inputs is now required. This list must begin with the appropriate symbol.

`Beginning of SECTION command not found`
: A new section was supposed to begin. This indicates that the previous section had extra data that was not processed.

`Beginning of SUBSECTION command not found`
: A new subsection was supposed to begin. This indicates that the previous subsection had extra data that was not processed.

`Beginning of SUBSUBSECTION command not found`
: A new sub-subsection was supposed to begin. This indicates that the previous sub-subsection had extra data that was not processed.

`Binary addition or subtraction in units expression`
: A units expression cannot involve addition or subtraction. Negation (or positivization) is allowed. (Override by enclosing the units expression in double square brackets, as in `[[1+1]]`).

`End of list symbol } not found`
: A user declared number of inputs was required. This list must end with the appropriate symbol. Perhaps there are too many items in the list.

`Error after input file <file name> line number <number>`
: An otherwise unspecified error occurred somewhere after the given line of the given file. The error may be in a subsequently included file.

`Error allocating memory on processor <number>`
: There was an exhaustion of memory (on the given processor). You can first try to reduce the amount of workspace memory asked for. If this fails, you may be able to reduce the size of the problem you are solving. Finally, you can try to use a machine with more memory.

`Error between input file <file name> line number <number> and input file <file name> line number <number>`
: An otherwise unspecified error occurred somewhere between the given lines of the given files. The error may be in an intervening included file.

`Error in data block`
: There is an error in a numerical data block (see §4.7.2 on page 22, and also grid array in §4.7.3 on page 22). Perhaps the data block has the wrong number of items.

`Error in input file <file name> before line number <number>`
: An otherwise unspecified error occurred somewhere before the given line of the given file. The error may be in a previously included file. This message may be preceded by a more specific error message.

`Error in input file <file name> on line number <number>`
: An otherwise unspecified error occurred on the given line of the given file. This message may be preceded by a more specific error message. The item read was not of the appropriate type to continue (e.g., an integer was expected, but a real number was found). The actual error could be well before this point in the file(s).

`Error on processor <number>`
: An unspecified error occurred on the given processor.

`Error opening input file <file name>`
: The given file was not able to be opened. Perhaps it is misspelled or there is an error in the directory tree path to the file.

`Expression has nonnested parentheses`
: A units expression cannot have nonnested parentheses.

`Grid array expected`
: A grid array (see §4.7.3 on page 22) was expected but not found. Perhaps a begin list symbol '{' was left out.

`Key not found: <list of words>`
: The key-word from the given list was not found. It is likely missing, misspelled, or out of order.

`Premature beginning of SECTION command found`
: A new section was begun before the last one was finished.

`Premature beginning of SUBSECTION command found`
: A new subsection was begun before the last one was finished.

`Premature beginning of SUBSUBSECTION command found`
: A new sub-subsection was begun before the last one was finished.

`Premature end of data`
: The data file ended before all appropriate data was read and processed.

`Section key not found: <list of words>`
: The section name from the given list was not found. It is likely missing, misspelled, or out of order.

`Sorry, there is an error in the computer code`
: Refer the problem to someone who can debug the code. This error message occurs generally only for errors that are relatively straightforward to resolve.

`Subsection key not found: <list of words>`
: The subsection name from the given list was not found. It is likely missing, misspelled, or out of order.

`Subsubsection key not found: <list of words>`
: The sub-subsection name from the given list was not found. It is likely missing, misspelled, or out of order.

`Too many nested include files: maximum <number> allowed`
: While there is no limit on the number of files that may be included, there is a maximum number that may be nested. Try to finish a file before calling for succeeding files, or have the code recompiled with a greater number of nesting levels allowed.

`Unit expression error`
: A units expression has an otherwise unspecified error.

`Unit expression has the wrong dimensions`
: A units expression has the wrong physical dimensions. (Override by enclosing the units expression in double square brackets, as in `[[2]]`)

`Units expression improperly formed or unknown unit`
: A units expression is improperly formed. It is possible that an unknown unit was given, and the expression became confused.

`Units expression is not positive`
: A units expression is not positive. (Override by enclosing the units expression in double square brackets, as in `[[-1]]`).

`Units expression is too long`
: A units expression is too long. You must shorten it, or work out the conversion factor yourself.

`Unknown $ command`
: The command name was misspelled or the desired command is not supported.

`Unknown unit`
: A units expression has an unknown unit, or perhaps it was misspelled. The command line argument `-u` can be used to obtain a list of supported units. New unit names and values can be added to the code and then it can be recompiled; otherwise, the numerical value of a unit (in the base units) can be used together with the double square bracket notation.

`Unmatched end comment command`
: An `$END_COMMENT` command appeared without a matching `$BEGIN_COMMENT` command.

`Unsupported input option`

The requested option is not supported. Try another option. To determine the supported options, use this manual (or perhaps the command line argument `-t` to display the type numbers of various options).

Chapter 7
The Output Data Files

The user may write output data files, as described in the chapter on input (Chapter 5). These files may be 3-D in space or 1-D in time.

The user may also write various other files for testing and debugging purposes, as described in the chapter on input.

7.1 Data that is 3-D in space

One can write 3-D files of the pressure, velocities, and concentrations. These files are formatted so that they can be viewed directly with the graphics package *Eye* or *Tecplot*, or the raw data is given, depending on the setting of `outFormat3D`.

These 3-D file names have the structure "data.number". Here, "data" is one of the following:

`pres` for pressure data;
`vel` for vector velocity data;
`velx` for x-velocity data;
`vely` for y-velocity data;
`velz` for z-velocity data;
`conc?` for concentration data of
the '?'*th* component (or a range of components).

There may be an appropriate graphics package extension appended to the file name. Only a single vector velocity file or 3 component velocity files will appear, as is appropriate for the graphics package. The "number" is sequential, and corresponds to the respective step number and time entry in the file `presTime`, `velTime`, or `concTime`, for pressure, velocity, or concentration data, respectively. The grid will be stored in the file `grid.000` if the graphics package supports a separate grid file; otherwise, the grid is stored in each individual output data file.

See the documentation on *Eye* and *Tecplot* for information on the format of these files. Both of these formats represent all data on the same grid, that consists of the centers of the cells (*not* the grid points!), the centers of the boundary faces, the centers of the boundary edges, and the eight corners of the domain. Thus, the output grid covers the same domain as the input grid; however, the input and output grids contain different points. Pressure and concentration data is cell-centered data, so in the output file, each cell center data value is associated to the appropriate point of the output grid (in the interior of the domain). On the boundary of the domain, the output grid points that are face centers of the input grid have associated with them the data value of the adjacent cell. The other output grid points are associated to averages of nearby boundary data values. This procedure is used so that no false maxima or minima are represented in the output files; however, this has the effect that any visual rendering may appear to be a bit too regular near the boundary. The velocities are not strictly cell-centered; for example, the x velocity `velX` is face centered in x

and cell-centered in y and z. Each cell centered value is obtained as the average of the two face centered values of the cell, and the boundary values are either given or derived as in the case of cell-centered data.

The *Eye* format is about as compact as possible. The grid is stored only once in the grid file, and each data file contains a single data item (pressure, velocity component, or species concentration). Thus, the storage is minimized, but the number of files is maximized.

The *raw data* format is as compact as possible. No grid is produced, and each data file contains a single data item (pressure, velocity component, or species concentration). The order follows the logically rectangular structure of the grid, and is given with the x coordinate varying most rapidly, then y, and finally z (as in Fortran). All files contain cell-centered data and are of size `(nx,ny,nz)`, except the velocity files, which contain face-centered data (e.g., the x velocity file is of size `(nx+1,ny,nz)`).

7.2 Data that is 1-D in time

One can write 1-D files of the concentration history of one or more species over a fixed region of space. These files are formatted so that they can be viewed directly with the graphics package *Tecplot*, as set by `outFormat1D`.

These 1-D file names begin with "`concHist`", have the number of the history, and then two letters describing the type of data recorded. More specifically, the names are one of

`concHist?CP` for the pore volume average of primary phase concentration of species m:

$$\sum_n \phi_n c_{m,n} V_n / \sum_n \phi_n V_n,$$

`concHist?CT` for the pore volume average of total moles (flowing and sorbed) of species m:

$$\sum_n (\phi_n + \alpha_m \sigma_{m,n}) c_{m,n} V_n / \sum_n \phi_n V_n,$$

`concHist?MP` for total primary phase moles of species m:

$$\sum_n \phi_n c_{m,n} V_n$$

`concHist?MT` for the total moles (flowing and sorbed) of species m:

$$\sum_n (\phi_n + \alpha_m \sigma_{m,n}) c_{m,n} V_n,$$

where the '?' is the history number, and V_n is the volume of the nth grid element of the fixed region. A final "`i,j,k`" or "`,i,j,k-i,j,k`" gives the cell-centered data index numbers of the fixed grid cell or range of cells (`histIndexLo_t_x`, `histIndexLo_t_y`, `histIndexLo_t_z` and, if different, `histIndexHi_t_x`, `histIndexHi_t_y`, `histIndexHi_t_z`).

See the documentation on *Tecplot* for information on the format of these files.

Appendix A
Sample Input Files

A.1 An input template

```
GENERAL INPUT DATA FILE TEMPLATE FOR PARSSIM1

This is a template of the input data file for PARSSIM1 version 2.1.

================================================================================

$S GENERAL INFO
0 || 1   # computeFlow: Compute flow
0 || 1 || 2 || 3   # computeTransport: Compute transport
0 || 1 || 2 || 3 || -1   # computeChemistry: Compute chemistry
0 || 1   # computeRND: Compute RND
[<units expression [L]>]   # baseLength: Internal length base units
[<units expression [M]>]   # baseMass: Internal mass base units
[<units expression [T]>]   # baseTime: Internal time base units
[degK] || [degC] || [degF]   # baseTemperature: [degK], [degC], or [degF]

$S SOLUTION PARAMETERS
<integer: -1> <integer: -1> <integer: 1>   # nDom_x, _y, _z:
      # Parallel subdomain divisions (-1=auto select)

$SS Flow
# This sectional unit is needed only if computeFlow != 0
<integer: 100>   # maxIterIF_f: Interface maximum number of iterations
<real [1]: 1.0e-6>   # relTolIF_f: Interface relative tolerance
<real [1]>   # absTolIF_f: Interface absolute tolerance
0 || 1 || 5 <: 5>    # pcTypeIF_f: Interface preconditioner
<integer: 48000>   # dWorkspace_f: Double workspace for flow

$SS Transport
# This sectional unit is needed only if computeTransport != 0
<integer: 100>   # maxIter_t: Dispersion maximum number of iterations
<real [1]: 1.0e-6>   # relTol_t: Dispersion relative tolerance
<integer: 10000>   # dWorkspace_t: Double workspace for transport
<real [1]: -1>   # cflFactor: Numerical CFL factor

$SSS CMM
# This sectional unit is needed only if computeTransport
# selects the CMM advection scheme.
<integer: 2>   # ntCutMax_t: Maximum number of time step cuts
<integer: 2> <integer: 2> <integer: 2>   # padx, pady, padz:
```

```
      # Number of cells for subdomain overlap (pad)
<real [1]: 10>   # volTol: Volume discrepancy tolerance
<integer: 1> <integer: 1> <integer: 1>
      # volRefine_t_x, volRefine_t_y, volRefine_t_z: Trace-back volume refinement

$SS Chemistry
# This sectional unit is needed only if computeChemistry > 0
0 || 1 <: 1>   # chmLoadBalFlag: Parallel chemistry load balancing
<integer: 100>   # chmMaxIter: Maximum number of nonlinear iterations
<real [1]>   # chmAbsTol: Absolute KKT tolerance
<real [moles/L^3]]: 1.0e-16>   # chmEpsConc: Minimal concentration parameter
0 || 1 <: 0>   # chmScaleFlag: Diagonal scaling
0 || 1 <: 0>   # chmTestSolFlag: Test second-order sufficiency conditions
-1 || 0 || 1 <: -1>   # chmGuessType: initial guess (-1=auto,0=transported,1=previous)
0 || 1 <: 0>   # chmInterpFlag: Use interpolation in the back-track line-search
<integer: 10>   # nViol: Number of non-monotonic line-searches
<real [1]: 1.0e-4>   # chmAlpha: Alpha parameter
0 || 1 <: 1>   # hessFlag: Analytical Hessian
<real [1]: 0.8>   # tauMin: Movement to the boundary factor
<real [1]: 1.0e-2>   # chmRho: IP reduction factor of perturbation parameter
<integer>   # ntReact: Number of reaction steps per transport step
0 || 1 || 2   # odeAlgType: ODE integration (0=RK1, 1=RK2, 2=RK4)
0 || 1 <: 0>   # switchFlag: Species switching
<integer: 1000>   # dWorkspace_c: Double workspace for chemistry
<integer: 1000>   # iWorkspace_c: Integer workspace for chemistry

$S TIME
<real [T]: 0>   # tInitial: Initial time
<real [T]>   # tFinal: Final time

$SS Flow
# This sectional unit is needed only if computeFlow != 0
<integer>   # nStepsMax_f: Maximum number of flow steps
<integer>   # ndtMax_f: Number of flow time step sizes
{
# For each of the ndtMax_f time step sizes, give the following
<real [T]>   # tdtMax_f: Time of onset
<real [T]>   # dtMax_f: Time step size
}

$SS Transport
# This sectional unit is needed only if computeTransport != 0,
# computeChemistry != 0, or computeRND != 0
<integer>   # nStepsMax_t: Maximum number of transport steps
<integer>   # nStepsMax_t_per_f: Maximum number of transport steps per flow step
<integer>   # ndtMax_t: Number of transport time step sizes
{
# For each of the ndtMax_t time step sizes, give the following
<real [T]>   # tdtMax_t: Time of onset
<real [T]>   # dtMax_t: Time step size
}

$S GRID
<integer> <integer> <integer>   # nx, ny, nz: Number of grid cells
```

```
<integer> <integer> <integer>   # periodicBC_x, _y, _z: Boolean for periodicity
0 || 1   # zIsDepth: Direction of the z coordinate
uniform || rectangular || xy-rectangular || {   # gridType:
      # uniform, rectangular, xy-rectangular, or {

# Use the following when gridType is uniform
<real [L]> <real [L]>   # xMin, xMax: Minimal and maximal x points
<real [L]> <real [L]>   # yMin, yMax: Minimal and maximal y points
<real [L]> <real [L]>   # zMin, zMax: Minimal and maximal z points

# Use the following when gridType is rectangular
{
# For each of the nx+1 x grid points, give the following
<real [L]>   # xp: The x grid points
}
{
# For each of the ny+1 y grid points, give the following
<real [L]>   # yp: The y grid points
}
{
# For each of the nz+1 z grid points, give the following
<real [L]>   # zp: The z grid points
}

# Use the following when gridType is xy-rectangular
{
# For each of the nx+1 x grid points, give the following
<real [L]>   # xp: The x grid points
}
{
# For each of the ny+1 y grid points, give the following
<real [L]>   # yp: The y grid points
}
{
# For each of the (nx+1)*(ny+1)*(nz+1) z grid points, give the following
<real of a data block [L]>   # zp: The z grid points
}

# Use the following when gridType is {
# For each of the (nx+1)*(ny+1)*(nz+1) x grid points, give:
<real of a data block [L]>   # xp: The x grid points
}
{
# For each of the (nx+1)*(ny+1)*(nz+1) y grid points, give the following
<real of a data block [L]>   # yp: The y grid points
}
{
# For each of the (nx+1)*(ny+1)*(nz+1) z grid points, give the following
<real of a data block [L]>   # zp: The z grid points
}

$S MATERIAL PROPERTIES
<real [L/T^2]: 9.8 [m/sec^2]>   # gravity: Gravitational constant
<real grid array [1]>   # porosity: Porosity
```

```
$SS Permeability || $SS Conductivity
# This sectional unit is needed only if computeFlow != 0
scalar || diagonal || symmetric || face   # permType:
      # scalar, diagonal, symmetric, or face

# Use the following when permType is scalar
<real grid array [L^2 (permeability) or L/T (conductivity)]>   # perm:
      # Scalar permeabilities

# Use the following when permType is diagonal
by cells || by components   # permGrouping:
      # Group by grid cells or by tensor components
xx yy zz || <any permutation>   # permComponentOrder:
      # Order of tensor's components
<1 or 3 real grid arrays [L^2 (permeability) or L/T (conductivity)]>   # perm:
      # Diagonal tensor permeabilities

# Use the following when permType is symmetric
cells || components   # permGrouping:
      # Group by cells or by tensor components
xx yy zz xy xz yz || <any permutation>   # permComponentOrder:
      # Order of tensor's components
<1 or 6 real grid arrays [L^2 (permeability) or L/T (conductivity)]>   # perm:
      # Symmetric tensor permeabilities

# Use the following when permType is face
<3 real grid arrays [L^2 (permeability) or L/T (conductivity)]>   # perm:
      # Face permeabilities

$SS Dispersion
# This sectional unit is needed only if computeTransport != 0
<integer>   # uniformDispersion: Flag for uniform dispersion

# Use this syntax if uniformDispersion is true (i.e., 1)
<real [L^2/T]>   # molDiff: Molecular diffusion
<real [L]>   # longDisp: Longitudinal dispersion
<real [L]>   # transDisp: Transverse dispersion

# Use this syntax if uniformDispersion is false (i.e., 0)
<real grid array [L^2/T]>   # molDiffAry: Molecular diffusion array
<real grid array [L]>   # longDispAry: Longitudinal dispersion array
<real grid array [L]>   # transDispAry: Transverse dispersion array

$SS Sorption
# This sectional unit is needed only if computeTransport != 0,
# computeChemistry != 0, or computeRND != 0
<integer>   # nSorpTypes: Number of sorption types
{
# For each of the nSorpTypes sorption types, give the following
<real grid array [1]> || porosity   # sorp: Sorption capacities
}

$S PHASE PROPERTIES
```

```
<real [M/(LT)]>   # fluidViscosity: Flowing phase viscosity
<real [M/L^3]>   # fluidDensity: Flowing phase mass density
<integer>   # nPhases: Number of phases

$SS Phase
# Include one such sectional unit per phase as specified by nPhases
<word of characters>   # phaseNames: Phase name
multi-species || single-species   # phaseType: multi-species or single-species
<real [1/L^3]> || <omit>   # chmPhaseDensity: Molar density

$S CHEMISTRY
# This sectional unit is needed only if computeChemistry > 0
<real [temperature reading]>   # absTemp: Equilibrium temperature
<real [M/(LT^2]>   # chemPres: Equilibrium pressure
ideal || non-ideal   # ideal: ideal or non-ideal

$S CHEMICAL SPECIES
<integer>   # nComps: Number of components
<integer>   # nProducts: Number of products
# nSpecies = nComps + nProducts

$SS Component || $SS H Component || $SS H2O Component || $SS Product
# Include one such sectional unit per component as specified by nComps
# Follow by one such sectional unit per product as specified by nProducts
<word of characters>   # specieName: Specie name
<real [M]>   # molecularWeight: Molecular weight
<real [1]>   # phaseDist: The alpha parameter in Henry's Law
<integer> || <omit>   # sorpType: Sorption type

$SSS Chemistry
# This sectional unit is needed only if computeChemistry > 0

# Use this syntax if the subsection heading is Component,
# H Component, or H2O Component
<integer>   # phaseIdentity: Participating phase
<real [1]>   # compCharge: Charge

# Use this syntax if the subsection heading is Product
<integer>   # phaseIdentity: Participating phase
{
# For each of the nComps components, give the following
<real [1]>   # stoich: Stoichiometry formula number
}
0 || 1 || 2   # reactionType: Reaction type

# Use this syntax if reactionType specifies an equilibrium reaction
<real [1]>   # pK: Log 10 equilibrium constant

# Use this syntax if reactionType specifies a mass-action type kinetic reaction
<real [1]> <real [1]>   # pKf, pKb: Log 10 forward and backward rate-constants
{
# For each of the nComps components, give the following
<real [1]>   # stoichK: Rate-law powers
}
```

```
# Use this syntax if reactionType specifies a monod type kinetic reaction
<real [1]> <real [1]: -99>   # pKf, pKb: Log 10 forward & backward rate-constants
{
# For each of the nComps components, give the following
<real [1]>   # stoichK: Rate-law powers
}
{
# For each of the nComps components, give the following
<real [1]>   # halfSatConst: Half saturation constants
}

$S MISCIBLE DISPLACEMENT
# This sectional unit is needed only if computeFlow != 0 and
# computeTransport != 0, computeChemistry != 0, or computeRND != 0
0 || 1   # modelMiscDisp: Miscible displacement model

# Use the following when modelMiscDisp selects
# the quarter power mixing rule
{
# For each of the nSpecies species, give the following
<real [1]>   # viscosityRatio: Viscosity ratio
}

$S RADIONUCLIDE DECAY
# This sectional unit is needed only if computeRND != 0
<integer>   # nChains: Number of RND chains

$SS Chain
# Include one such sectional unit per chain as specified by nChains
<integer>   # chainLen: Length of chain
linear || branching   # chainType: linear or branching
{
# For each of the chainLen species, give the following
<integer>   # specieNumber: Specie number
<real [T]>   # halfLife: Half-life
<omit> || <real [1]> ...   # branchRatios: Branch ratios
}

$S SPECIALIZED REACTIONS
# This sectional unit is needed only if computeChemistry < 0
<integer>   # rxn_nComps: Number of component species in reactions
{
# For each of the rxn_nComps component species, give the following
<integer>   # rxn_comp: Species number of the next reaction component
}
<integer>   # rxn_nMicroSteps: Number of reaction micro steps
<double [1]>   # rxn_tolerance: Convergence tolerance parameter
<integer>   # rxn_nDParams: Number of real parameters for user's rxn fcns
{
# For each of the rxn_nDParams real parameters, give the following
<real [1]>   # rxn_dParams: Real parameter
}
<integer>   # rxn_nIParams: Number of integer parameters for user's rxn fcns
```

```
{
# For each of the rxn_nIParams integer parameters, give the following
<integer>   # rxn_iParams: Integer parameter
}

$S INITIAL CONDITIONS

$SS Flow
# This sectional unit is needed only if computeFlow = 0
<real grid array [L/T]>   # velX: X velocity
<real grid array [L/T]>   # velY: Y velocity
<real grid array [L/T]>   # velZ: Z velocity

$SS Transport
# This sectional unit is needed only if computeTransport != 0,
# computeChemistry != 0, or computeRND != 0
{
# For each of the nSpecies species, give the following
<real grid array [1/L^3]>   # conc: Molar concentration
}

$S BOUNDARY CONDITIONS
<integer>   # nBCRegions: Number of boundary regions
<integer>   # maxnBCInterp_f: Maximum number of flow interpolation times
<integer>   # maxnBCInterp_t: Maximum number of transport interpolation times
# The following lines appear only if periodicBC_x is 0
<integer grid array>   # bcRegion_x_min: Boundary region map
<integer grid array>   # bcRegion_x_max: Boundary region map
# The following lines appear only if periodicBC_y is 0
<integer grid array>   # bcRegion_y_min: Boundary region map
<integer grid array>   # bcRegion_y_max: Boundary region map
# The following lines appear only if periodicBC_z is 0
<integer grid array>   # bcRegion_z_min: Boundary region map
<integer grid array>   # bcRegion_z_max: Boundary region map

$SS Region
# Include one such sectional unit per boundary region as specified by nBCRegions

$SSS Flow
# This sectional unit is needed only if computeFlow != 0
0 || 1 || 2   # bcType_f: BC type

# The following lines appear only if a function of time
# needs to be specified; that is, bcType_f != 0
<omit> || pressure || potential || head || hydraulicHead   # bcPresType_in:
      # Dirichlet pressure, potential, head, or hydraulicHead
<integer>   # nBCInterp_f: Number of interpolation times
{
# For each of the nBCInterp_f interpolation times, give the following
<real [T]>   # bc_f_time: BC time
<real [L/T (Neumann) or M/(L*T**2) (Dir pres/pot) or L (Dir head/hHead)]>
      # bc_f_value: BC value
}
```

```
$SSS Transport
# This sectional unit is needed only if computeTransport != 0
0 || 1 || 2   # bcType_t: BC type

# The following lines appear only if a function of time
# needs to be specified; that is, bcType_t != 0
{
# For each of the nSpecies species, give the following
<integer>   # nBCInterp_t: Number of interpolation times
{
# For each of the nBCInterp_t interpolation times, give the following
<real [T]>   # bc_t_time: BC time
<real [1/L^3]>   # bc_t_value: BC conc value
}
}

$S WELLS
<integer>   # nWells: Number of wells
<integer>   # maxnWellInterp_f: Maximum number of flow interpolation times
<integer>   # maxnWellInterp_t: Maximum number of transport interpolation times

$SS Well
# Include one such sectional unit per well as specified by nWells
0 || 1 || 2 || 3 || 4   # wellType: Well type

# If the well is not inactive, include the following information
<real [L]>   # wellRadius: Well radius
<integer> <integer>   # hWellIndex: The x and y indices of well
<integer>   # zWellIndexLo: Lower z index of well
<integer>   # zWellIndexHi: Higher z index of well
<omit> || <integer>   # zWellIndexLo: Lower z index of well
<omit> || <integer>   # zWellIndexHi: Higher z index of well

$SSS Flow
# This sectional unit is needed only if computeFlow != 0

# Use if wellType selects a bottomhole producer
<real [M/(L*T^2)]>   # wellPres: Well pressure potential

# The following lines appear only if wellType does not select a bottomhole producer
<integer>   # nWellInterp_f: Number of interpolation times
{
# For each of the nWellInterp_f interpolation times, give the following
<real [T]>   # wellInj_f_time: Well time
<real [L^3/T]>   # wellInj_f_value: Well injection or extraction rate
}

$SSS Transport
# This sectional unit is needed only if computeTransport != 0 and
# wellType does not select an extractor or a bottomhole producer
{
# For each of the nSpecies species, give the following
<integer>   # nWellInterp_t: Number of interpolation times
{
```

```
# For each of the nWellInterp_t interpolation times, give the following
<real [T]>   # wellInj_t_time: Well time
<real [1/L^3]>   # wellInj_t_value: Well conc value
}
}

$S LEAKS
<integer>   # nLeaks: Number of leaks
<integer>   # maxnLeakInterp: Maximum number of interpolation times

$SS Leak
# Include one such sectional unit per leak as specified by nLeaks
<integer> <integer> <integer>   # leakIndexLo: Lower cell of leak
<integer> <integer> <integer>   # leakIndexHi: Upper cell of leak
{
# For each of the nSpecies species, give the following
<integer>   # nLeakInterp: Number of interpolation times
{
# For each of the nLeakInterp interpolation times, give the following
<real [T]>   # leakInj_t_time: Leak time
<real [1/T]>   # leakInj_t_value: Leak conc value
}
}

$S OUTPUT
<integer>   # runNumber: Run number
<line of characters describing the run>
      # runDescription: Title or description
0 || 1   # verbosity: Driver verbosity flag
0 || 1 || 2 || 3 || 4   # debug: Driver debug flag
<word of characters>   # outDir: Output directory
0 || 1   # outFormat3D: Output 3-D file format
<integer>   # nSpeciesPerOutfile: Number of species per 3-D output file
0 || 1   # initialOut: Output 3-D initial conditions flag
1   # outFormat1D: Output 1-D file format

$SS Flow
# This sectional unit is needed only if computeFlow != 0
<integer>   # dStepOut_pres: Steps between pressure outputs
<integer>   # dStepOut_vel: Steps between velocity outputs
<real [T]>   # dtOut_pres: Time between pressure outputs
<real [T]>   # dtOut_vel: Time between velocity outputs
potential || pressure || hydraulicHead || head   # presType_out:
      # potential, pressure, hydraulicHead, or head
<integer 0-1000>   # outFlags_f_1: Level of flow verbosity
<integer 0-1000>   # outFlags_f_2: Level of flow debugging

$SS Transport
# This sectional unit is needed only if computeTransport != 0,
# computeChemistry != 0, or computeRND != 0
<integer>   # dStepOut_conc: Steps between concentration outputs
<real [T]>   # dtOut_conc: Time between concentration outputs
<integer>   # nHistories_t: Number of 1-D time histories
0 || 1   # outFlags_t_1: Monitor transport progress
```

```
0 || 1   # outFlags_t_2: Monitor transport time step cuts
0 || 1   # outFlags_t_3: Monitor transport dispersion solver
<integer>   # outFlags_t_4: Debug transport CMM trace-back points
0 || 1   # outFlags_t_5: Debug (write) transport conc
0 || 1   # outFlags_t_6: Debug transport call and set-up
0 || 1   # outFlags_t_7: Debug transport CMM trace-back integrals.
<integer 0-4>   # outFlags_t_8: Debug chemistry
0 || 1   # outFlags_t_9: Debug transport dispersion linear system assembly

$SSS History
# Include one such sectional unit per history as specified by nHistories_t
<integer>   # histType_t: History type (1=CP,2=CT,3=MP,4=MT)
<integer> <integer> <integer>   # histIndexLo_t_x, _y, _z: Lower cell of region
<integer> <integer> <integer>   # histIndexHi_t_x, _y, _z: Upper cell of region
<integer>   # nHistSpecies_t: Number of species in the history
{
# For each of the nHistSpecies_t species, give the following
<integer>   # histSpecie_t: Species in the history
}
<integer>   # dStepOut_hist_t: Steps between history outputs
<real [T]>   # dtOut_hist_t: Time between history outputs
```

A.2 Sample input file 1

```
SAMPLE GENERAL INPUT FILE FOR PARSSIM1 version 2.1

================================================================================

$S GENERAL INFO
1   # computeFlow: Compute flow
2   # computeTransport: Compute transport
1   # computeChemistry: Compute chemistry
0   # computeRND: Compute RND
[cm]    # baseLength: Internal length base units
[g]     # baseMass: Internal mass base units
[min]   # baseTime: Internal time base units
[degC]  # baseTemperature: [degK], [degC], or [degF]

$S SOLUTION PARAMETERS
-1, -1, -1  # nDom_x, _y, _z: Parallel subdomain divisions (-1=auto select)

$SS Flow
100     # maxIterIF_f: Interface maximum number of iterations
1.0e-6  # relTolIF_f: Interface relative tolerance
0       # absTolIF_f: Interface absolute tolerance
5       # pcTypeIF_f: Interface preconditioner
100000  # dWorkspace_f: Double workspace for flow

$SS Transport
100     # maxIter_t: Dispersion maximum number of iterations
1.0e-6  # relTol_t: Dispersion relative tolerance
10000   # dWorkspace_t: Double workspace for transport
-1      # cflFactor: Numerical CFL factor
```

```
$SSS CMM
2     # ntCutMax_t: Maximum number of time step cuts
2,2,2 # padx, pady, padz: Number of cells for subdomain overlap (pad)
10    # volTol: Volume discrepancy tolerance
1,1,1 # volRefine_t_x,volRefine_t_y,volRefine_t_z: Trace-back volume refinement

$SS Chemistry
0        # chmLoadBalFlag: Parallel chemistry load balancing
100      # chmMaxIter: Maximum number of nonlinear iterations
.0001    # chmAbsTol: Absolute KKT tolerance
1.0e-16  # chmEpsConc: Minimal concentration parameter
0        # chmScaleFlag: Diagonal scaling
0        # chmTestSolFlag: Test second-order sufficiency conditions
-1       # chmGuessType: initial guess (-1=auto,0=transported,1=previous)
0        # chmInterpFlag: Use interpolation in the back-track line-search
10       # nViol: Number of non-monotonic line-searches
1.0e-4   # chmAlpha: Alpha parameter
1        # hessFlag: Analytical Hessian
0.8      # tauMin: Movement to the boundary factor
1.0e-2   # chmRho: IP reduction factor of perturbation parameter
2        # ntReact: Number of reaction steps per transport step
2        # odeAlgType: ODE integration (0=RK1, 1=RK2, 2=RK4)
0        # switchFlag: Species switching
1000     # dWorkspace_c: Double workspace for chemistry
1000     # iWorkspace_c: Integer workspace for chemistry

$S TIME
0        # tInitial: Initial time
30[min]  # tFinal: Final time

$SS Flow
1   # nStepsMax_f: Maximum number of flow steps
1   # ndtMax_f: Number of flow time step sizes
{
0       # tdtMax_f: Time of onset
1[hr]   # dtMax_f:  Time step size
}

$SS Transport
2   # nStepsMax_t: Maximum number of transport steps
-1  # nStepsMax_t_per_f: Maximum number of transport steps per flow step
1   # ndtMax_t: Number of transport time step sizes
{
0,      # tdtMax_t: Time of onset
30[s]   # dtMax_t:  Time step size
}

$S GRID
8,4,4   # nx, ny, nz: Number of grid cells
0,0,0   # periodicBC_x, _y, _z: Boolean for periodicity
1       # zIsDepth: Direction of the z coordinate
uniform # gridType: uniform, rectangular, xy-rectangular, or {
0,  18[m]   # xMin, xMax: Minimal and maximal x points
```

```
0,  12[m]   # yMin, yMax: Minimal and maximal y points
1, 100[cm]  # zMin, zMax: Minimal and maximal z points

$S MATERIAL PROPERTIES
9.8 [m/sec^2]      # gravity: Gravitational constant
{ 64@0.25 64@0.5 } # porosity: Porosity

$SS Permeability
diagonal  # permType: scalar, diagonal, or symmetric

by components  # permGrouping: Group by grid cells or by tensor components
xx yy zz       # permComponentOrder: Order of tensor's components
# perm: Diagonal tensor permeabilities
{[md] 40 45 47 43, 40 45 47 43,
      39 36  2@38, 39 36  2@38,
      39 43 41 42, 39 43 41 42,
      45 47 55 65; 45 47 55 65,
      50 55 57 53, 50 55 57 53,
      49 46  2@48, 49 46  2@48,
      49 53 51 52, 49 53 51 52,
      55 57 65 75, 55 57 65 75,
      40 45 47 43, 40 45 47 43,
      39 36  2@38, 39 36  2@38,
      39 43 41 42, 39 43 41 42,
      45 47 55 65; 45 47 55 65,
      50 55 57 53, 50 55 57 53,
      49 46  2@48, 49 46  2@48,
      49 53 51 52, 49 53 51 52,
      55 57 65 75, 55 57 65 75
}
constant 40[md]
nearly constant 10[md]
 1 {1:4, 1:4, 1:1; 20[md]}

$SS Dispersion
1           # uniformDispersion: Flag for nonuniform dispersion
1[cm^2/day] # molDiff: Molecular diffusion
10[cm]      # longDisp: Longitudinal dispersion
1[cm]       # transDisp: Transverse dispersion

$SS Sorption
2   # nSorpTypes: Number of sorption types
{
porosity        # sorp: Sorption capacities
constant 50[%]
}

$S PHASE PROPERTIES
2[cp]        # fluidViscosity: Flowing phase viscosity
0.9[g/cm^3]  # fluidDensity: Flowing phase mass density
2            # nPhases: Number of phases

$SS Phase #1
OIL_PHASE       # phaseNames: Phase name
```

```
multi-species   # phaseType: multi-species or single-species
1[/cc]          # chmPhaseDensity: Molar density

$SS Phase #2
ROCK            # phaseNames: Phase name
single-species  # phaseType: multi-species or single-species

$S CHEMISTRY
120[degF]  # absTemp: Equilibrium temperature
3.5[atm]   # chemPres: Equilibrium pressure
ideal      # ideal: ideal or non-ideal

$S CHEMICAL SPECIES
2   # nComps: Number of components
1   # nProducts: Number of products

$SS H Component #1
H+   # specieName: Specie name
1[g] # molecularWeight: Molecular weight
.1   # phaseDist: The alpha parameter in Henry's Law
1    # sorpType: Sorption type

$SSS Chemistry
1   # phaseIdentity: Participating phase
1   # compCharge: Charge

$SS Component #2
Oil   # specieName: Specie name
90[g] # molecularWeight: Molecular weight
.2    # phaseDist: The alpha parameter in Henry's Law
2     # sorpType: Sorption type

$SSS Chemistry
1   # phaseIdentity: Participating phase
0   # compCharge: Charge

$SS Product #3
Mineral # specieName: Specie name
94[g]   # molecularWeight: Molecular weight
-1      # phaseDist: The alpha parameter in Henry's Law

$SSS Chemistry
2     # phaseIdentity: Participating phase
{
4  1  # stoich: Stoichiometry formula number
}
1          # reactionType: Reaction type
3.30 -5.0  # pKf, pKb: Log 10 forward and backward rate-constants
{
0.0  1.    # stoichK: Rate-law powers
}

$S MISCIBLE DISPLACEMENT
1        # modelMiscDisp: Miscible displacement model
```

```
{
.8 1 -1  # viscosityRatio: Viscosity ratio
}

$S INITIAL CONDITIONS

$SS Flow # This sectional unit is needed only if computeFlow = 0
constant 10[ft/day]   # velX: X velocity
constant 0            # velY: Y velocity
constant 0            # velZ: Z velocity

$SS Transport
{
constant .1[/cc]  # conc: Molar concentration
constant .2[/cc]
constant .2[/cc]
}

$S BOUNDARY CONDITIONS
3   # nBCRegions: Number of boundary regions
1   # maxnBCInterp_f: Maximum number of flow interpolation times
2   # maxnBCInterp_t: Maximum number of transport interpolation times
constant 1   # bcRegion_x_min: Boundary region map
constant 2   # bcRegion_x_max: Boundary region map
constant 3   # bcRegion_y_min: Boundary region map
constant 3   # bcRegion_y_max: Boundary region map
constant 3   # bcRegion_z_min: Boundary region map
constant 3   # bcRegion_z_max: Boundary region map

$SS Region #1

$SSS Flow
2         # bcType_f: BC type
potential # bcPresType_in: Dirichlet pressure,potential,head, or hydraulicHead
1         # nBCInterp_f: Number of interpolation times
{
0         # bc_f_time:  BC time
4[atm]    # bc_f_value: BC value
}

$SSS Transport
1   # bcType_t: BC type
{
# nBCInterp_t: Number of interpolation times
# bc_t_time, bc_t_value: BC time, BC conc value
1 {0, 0.2}
2 {1[yr], 0 ; 1[yr], 0.4}
1 {0, 0}
}

$SS Region #2

$SSS Flow
2         # bcType_f: BC type
```

```
potential # bcPresType_in: Dirichlet pressure,potential,head, or hydraulicHead
1         # nBCInterp_f: Number of interpolation times
{
0         # bc_f_time:  BC time
3[atm]    # bc_f_value: BC value
}

$SSS Transport
0   # bcType_t: BC type

$SS Region #3

$SSS Flow
0   # bcType_f: BC type

$SSS Transport
0   # bcType_t: BC type

$S WELLS
2   # nWells: Number of wells
1   # maxnWellInterp_f: Maximum number of flow interpolation times
2   # maxnWellInterp_t: Maximum number of transport interpolation times

$SS Well #1
1      # wellType: Well type
10[cm] # wellRadius: Well radius
2,3    # hWellIndex: The x and y indices of well
3      # zWellIndexLo: Lower z index of well
4      # zWellIndexHi: Higher z index of well

$SSS Flow
1   # nWellInterp_f: Number of interpolation times
{
0             # wellInj_f_time:  Well time
10[USgal/hr]  # wellInj_f_value: Well injection or extraction rate
}

$SSS Transport
{
# nWellInterp_t: Number of interpolation times
# wellInj_t_time, wellInj_t_value: Well time, Well conc value
2 {0, 0 ; 30[days], 0.2[/cc]}
1 {0,0}
1 {0,0}
}

$SS Well #2
2      # wellType: Well type
15[cm] # wellRadius: Well radius
3,4    # hWellIndex: The x and y indices of well
1      # zWellIndexLo: Lower z index of well
2      # zWellIndexHi: Higher z index of well

$SSS Flow
```

```
1   # nWellInterp_f: Number of interpolation times
{
0             # wellInj_f_time:  Well time
10[USgal/hr]  # wellInj_f_value: Well injection or extraction rate
}

$S LEAKS
0   # nLeaks: Number of leaks
0   # maxnLeakInterp: Maximum number of interpolation times

$S OUTPUT
1     # runNumber: Run number
Sample data input file
      # runDescription: Title or description
1     # verbosity: Driver verbosity flag
0     # debug: Driver debug flag
.     # outDir: Output directory
1     # outFormat3D: Output 3-D file format
2     # nSpeciesPerOutfile: Number of species per 3-D output file
1     # initialOut: Output 3-D initial conditions flag
1     # outFormat1D: Output 1-D file format

$SS Flow
1          # dStepOut_pres: Steps between pressure outputs
1          # dStepOut_vel: Steps between velocity outputs
-1         # dtOut_pres: Time between pressure outputs
-1         # dtOut_vel: Time between velocity outputs
potential  # presType_out: potential, pressure, hydraulicHead, or head
0          # outFlags_f_1: Level of flow verbosity
0          # outFlags_f_2: Level of flow debugging

$SS Transport
1   # dStepOut_conc: Steps between concentration outputs
-1  # dtOut_conc: Time between concentration outputs
3   # nHistories_t: Number of 1-D time histories
1   # outFlags_t_1: Monitor transport progress
1   # outFlags_t_2: Monitor transport time step cuts
1   # outFlags_t_3: Monitor transport dispersion solver
0   # outFlags_t_4: Debug transport CMM trace-back points
0   # outFlags_t_5: Debug (write) transport conc
0   # outFlags_t_6: Debug transport call and set-up
0   # outFlags_t_7: Debug transport CMM trace-back integrals.
0   # outFlags_t_8: Debug chemistry
0   # outFlags_t_9: Debug transport dispersion linear system assembly

$SSS History #1
1      # histType_t: History type (1=CP,2=CT,3=MP,4=MT)
5,4,1  # histIndexLo_t_x, _y, _z: Lower cell of region
5,4,1  # histIndexHi_t_x, _y, _z: Upper cell of region
3      # nHistSpecies_t: Number of species in the history
{
1,3,2  # histSpecie_t: Species in the history
}
-1     # dStepOut_hist_t: Steps between history outputs
```

```
.4     # dtOut_hist_t: Time between history outputs

$SSS History #2
1      # histType_t: History type (1=CP,2=CT,3=MP,4=MT)
6,2,4  # histIndexLo_t_x, _y, _z: Lower cell of region
6,2,4  # histIndexHi_t_x, _y, _z: Upper cell of region
1      # nHistSpecies_t: Number of species in the history
{
1      # histSpecie_t: Species in the history
}
1      # dStepOut_hist_t: Steps between history outputs
-1     # dtOut_hist_t: Time between history outputs

$SSS History #3
1      # histType_t: History type (1=CP,2=CT,3=MP,4=MT)
2,2,4  # histIndexLo_t_x, _y, _z: Lower cell of region
2,2,4  # histIndexHi_t_x, _y, _z: Upper cell of region
2      # nHistSpecies_t: Number of species in the history
{
1,2    # histSpecie_t: Species in the history
}
2      # dStepOut_hist_t: Steps between history outputs
-1     # dtOut_hist_t: Time between history outputs
```

A.3 Sample input file 2

A.3.1 The main input file

```
SAMPLE GENERAL INPUT FILE FOR PARSSIM1 version 2.1

================================================================================
$S GENERAL INFO
1 # computFlow
2 # computeTransport
1 # computeChemistry
0 # computeRND
[m]     # baseLength: Internal length base units
[g]     # baseMass: Internal mass base units
[hr]    # baseTime: Internal time base units
[degC]  # baseTemperature: [degK], [degC], or [degF]

$S SOLUTION PARAMETERS
-1 -1 -1   # nDom_x, _y, _z: Parallel subdomain divisions (-1=auto select)
$SS Flow
100     # maxIterIF_f: Interface maximum number of iterations
1.0e-6  # relTolIF_f: Interface relative tolerance
1.0e-12 # absTolIF_f: Interface absolute tolerance
5       # pcTypeIF_f: Interface preconditioner
2048000 # dWorkspace_f: Double workspace for flow

$SS Transport
100      # maxIter_t: Dispersion maximum number of iterations
1.0e-6   # relTol_t: Dispersion relative tolerance
204800   # dWorkspace_t: Double workspace for transport
```

```
-1       # cflFactor: Numerical CFL factor

$SS Chemistry
0       # chmLoadBalFlag: Parallel chemistry load balancing
100     # chmMaxIter: Maximum number of nonlinear iterations
1.0E-09 # chmAbsTol: Absolute KKT tolerance
1.0e-30 # chmEpsConc: Minimal concentration parameter
0       # chmScaleFlag: Diagonal scaling
0       # chmTestSolFlag: Test second-order sufficiency conditions
-1      # chmGuessType: initial guess (-1=auto,0=transported,1=previous)
0       # chmInterpFlag: Use interpolation in the back-track line-search
10      # nViol: Number of non-monotonic line-searches
1.0e-4  # chmAlpha: Alpha parameter
1       # hessFlag: Analytical Hessian
0.995   # tauMin: Movement to the boundary factor
1.0e-3  # chmRho: IP reduction factor of perturbation parameter
1       # ntReact: Number of reaction steps per transport step
2       # odeAlgType: ODE integration (0=RK1, 1=RK2, 2=RK4)
0       # switchFlag: Species switching
10000   # dWorkspace_c: Double workspace for chemistry
10000   # iWorkspace_c: Integer workspace for chemistry

$S TIME
0        # tInitial: Initial time
.01 # 2. # tFinal: Final time

$SS Flow
100000 # nStepsMax_f: Maximum number of flow steps
1 { 0.0, 0.1 } # tdtMax_f, dtMax_f: Time of onset, Time step size

$SS Transport
100000 # nStepsMax_t: Maximum number of transport steps
-1     # nStepsMax_t_per_f: Maximum number of transport steps per flow step
2{0.0,0.000624 ; 1.0,0.005} # tdtMax_t, dtMax_t: Time of onset, Time step size

$S GRID
40 40 1 # nx, ny, nz: Number of grid cells
0,0,0   # periodicBC_x, _y, _z: Boolean for periodicity
1       # zIsDepth: Direction of the z coordinate
uniform
0.0  1.5   # xMin, xMax: Minimal and maximal x points
0.0  1.0   # yMin, yMax: Minimal and maximal y points
0.0  1.0   # zMin, zMax: Minimal and maximal z points

$S MATERIAL PROPERTIES
9.8 [m/sec^2]   # gravity: Gravitational constant
constant  1.0   # porosity: Porosity

$SS Permeability
scalar { $IGNORE_LINES 2 $INCLUDE perm.dat }

$SS Dispersion
1   # uniformDispersion: Flag for nonuniform dispersion
0.0 # molDiff: Molecular diffusion
```

```
0.0 # longDisp: Longitudinal dispersion
0.0 # transDisp: Transverse dispersion

$SS Sorption
1 {  porosity } # sorp: Sorption capacities for each type

$S PHASE PROPERTIES
1.2037e-8   # fluidViscosity: Flowing phase viscosity
1.0         # fluidDensity: Flowing phase mass density
4           # nPhases: Number of phases

$SS Phase # 1
Aqueous        # phaseNames: Phase name
multi-species  # phaseType: multi-species or single-species
55.55          # chmPhaseDensity: Molar density

$SS Phase # 2
Mineral1       # phaseNames: Phase name
single-species # phaseType: multi-species or single-species

$SS Phase # 3
Mineral2       # phaseNames: Phase name
single-species # phaseType: multi-species or single-species

$SS Phase # 4
Mineral3       # phaseNames: Phase name
single-species # phaseType: multi-species or single-species

$S CHEMISTRY
298.0 # absTemp: Equilibrium temperature
1.0   # chemPres: Equilibrium pressure
ideal # ideal: ideal or non-ideal

$S CHEMICAL SPECIES
4 # nComps: Number of components
3 # nProducts: Number of products

$SS Component #1
A+    # specieName: Specie name
100.0 # molecularWeight: Molecular weight
0.0   # phaseDist: The alpha parameter in Henry's Law
$SSS Chemistry
1     # phaseIdentity: Participating phase
+1    # compCharge: Charge

$SS Component #2
B-    # specieName: Specie name
100.0 # molecularWeight: Molecular weight
0.0   # phaseDist: The alpha parameter in Henry's Law
$SSS Chemistry
1     # phaseIdentity: Participating phase
-1    # compCharge: Charge

$SS Component #3
```

```
C+    # specieName: Specie name
200.0 # molecularWeight: Molecular weight
0.0   # phaseDist: The alpha parameter in Henry's Law
$SSS Chemistry
1     # phaseIdentity: Participating phase
+1    # compCharge: Charge

$SS Component #4
D-    # specieName: Specie name
200.0 # molecularWeight: Molecular weight
0.0   # phaseDist: The alpha parameter in Henry's Law
$SSS Chemistry
1     # phaseIdentity: Participating phase
-1    # compCharge: Charge

$SS Product #1
AB(s) # specieName: Specie name
200   # molecularWeight: Molecular weight
-99   # phaseDist: The alpha parameter in Henry's Law
$SSS Chemistry
2     # phaseIdentity: Participating phase
{ 1.0 1.0 0.0 0.0 } # stoich
0     # Equilibrium
0.0   # pK: Log 10 equilibrium constant

$SS Product #2
AC(s) # specieName: Specie name
300   # molecularWeight: Molecular weight
-99   # phaseDist: The alpha parameter in Henry's Law
$SSS Chemistry
3     # phaseIdentity: Participating phase
{ 1.0 0.0 1.0 0.0 } # stoich
0     # Equilibrium
-3.0103e-1 # pK: Log 10 equilibrium constant

$SS Product #3
DB(s) # specieName: Specie name
300   # molecularWeight: Molecular weight
-99   # phaseDist: The alpha parameter in Henry's Law
$SSS Chemistry
4     # phaseIdentity: Participating phase
{ 0.0 1.0 0.0 1.0 } # stoich
0     # Equilibrium
3.0103e-1 # pK: Log 10 equilibrium constant

$S MISCIBLE DISPLACEMENT 0 # None

$S INITIAL CONDITIONS

$SS Transport
{
  {$INCLUDE AB_IC.dat } # A+
  constant  1.0         # B-
  constant  1.0e-6      # C+
```

```
  constant  1.0e-6      # D-
  {$INCLUDE AB_IC.dat } # AB(s)
  {$INCLUDE AB_IC.dat } # AC(s)
  {$INCLUDE AB_IC.dat } # DB(s)
}

$S BOUNDARY CONDITIONS
3 # nBCRegions: Number of boundary regions
1 # maxnBCInterp_f: Maximum number of flow interpolation times
3 # maxnBCInterp_t: Maximum number of transport interpolation times
constant 3 # bcRegion_x_min: Boundary region map
constant 3 # bcRegion_x_max: Boundary region map
constant 3 # bcRegion_y_min: Boundary region map
constant 2 # bcRegion_y_max: Boundary region map
constant 3 # bcRegion_z_min: Boundary region map
constant 3 # bcRegion_z_max: Boundary region map

$SS Region # 1
$SSS Flow      0 # Noflow
$SSS Transport 1 # Inflow
{
  1 { 0.0  0.25 } # A+
  1 { 0.0  0.0  } # B-
  1 { 0.0  2.0  } # C+
  1 { 0.0  1.75 } # D-
  1 { 0.0  0.0  } # AB(s)
  1 { 0.0  0.0  } # AC(s)
  1 { 0.0  0.0  } # DB(s)
}

$SS Region # 2
$SSS Flow      1 # Neumann
1 { 0.0  -1.0}
$SSS Transport 0 # Noflow/outflow

$SS Region # 3
$SSS Flow      0 # Noflow
$SSS Transport 0 # Noflow/outflow

$S WELLS
1 # nWells: Number of wells
1 # maxnWellInterp_f: Maximum number of flow interpolation times
1 # maxnWellInterp_t: Maximum number of transport interpolation times

$SS Well #1
1    # injector
0.05 # radius
3,3  # x and y indices
1    # low z
1    # high z
$SSS Flow
1 { 0.0   1.5 }
$SSS Transport
{
```

```
 1 { 0.0  0.25 } # A+
 1 { 0.0  0.0  } # B-
 1 { 0.0  2.0  } # C+
 1 { 0.0  1.75 } # D-
 1 { 0.0  0.0  } # AB(s)
 1 { 0.0  0.0  } # AC(s)
 1 { 0.0  0.0  } # DB(s)
}

$S LEAKS
0   # nLeaks: Number of leaks
0   # maxnLeakInterp: Maximum number of interpolation times

$S OUTPUT
1 # runNumber: Run number
Fredrik 2Dtest problem
    # runDescription: Title or description
1   # verbosity: Driver verbosity flag
1   # debug: Driver debug flag
.   # outDir: Output directory
1   # outFormat3D: Output 3-D file format
10  # nSpeciesPerOutfile: Number of species per 3-D output file
1   # initialOut: Output 3-D initial conditions flag
1   # outFormat1D: Output 1-D file format

$SS Flow
-1 # dStepOut_pres: Steps between pressure outputs
-1 # dStepOut_vel: Steps between velocity outputs
-1 # dtOut_pres: Time between pressure outputs
-1 # dtOut_vel: Time between velocity outputs
potential # presType_out: potential, pressure, hydraulicHead, or head
0  # outFlags_f_1: Level of flow verbosity
0  # outFlags_f_2: Level of flow debugging

$SS Transport
-1    # dStepOut_conc: Steps between concentration outputs
.0045 # dtOut_conc: Time between concentration outputs
0     # nHistories_t: Number of 1-D time histories
1 # outFlags_t_1: Monitor transport progress
1 # outFlags_t_2: Monitor transport time step cuts
0 # outFlags_t_3: Monitor transport dispersion solver
0 # outFlags_t_4: Debug transport CMM trace-back points
0 # outFlags_t_5: Debug (write) transport conc
0 # outFlags_t_6: Debug transport call and set-up
0 # outFlags_t_7: Debug transport CMM trace-back integrals.
0 # outFlags_t_8: Debug chemistry
0 # outFlags_t_9: Debug transport dispersion linear system assembly
```

A.3.2 Auxilliary file "perm.dat"

```
A 10-layered medium
40 40 1
160@.1
160@.3
```

```
160@.5
160@.1
160@.5
160@.24
160@.3
160@.24
160@.5
160@.24
```

A.3.3 Auxilliary file “AB_IC.dat”

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Authors and Acknowledgments

Parssim1 was developed in the Center for Subsurface Modeling. Many people contributed to its development. The primary authors are acknowledged below.

Driver:
The driver and overall code framework was developed primarily by Todd Arbogast. Contributions were made also by Ashokkumar Chilakapati, Lawrence C. Cowsar, Robert McLay, Douglas Moore, and Ivan Yotov.

Flow:
Parcel [14] was developed initially by Lawrence Cowsar, and modified by Fredrik Saaf, Carol San Soucie, Ivan Yotov, and Robert McLay. Frederic d'Hennezel and Todd Arbogast developed the calling and set-up routines for flow. Joe Eaton developed the miscible displacement option.

Transport:
ParTrans was developed primarily by Todd Arbogast, Ashokkumar Chilakapati, and Douglas Moore. Other authors include Lawrence C. Cowsar, Clint N. Dawson, Frederic d'Hennezel, Mary F. Wheeler, and Ivan Yotov.

Chemistry:
The general geo-chemistry package was developed by Fredrik Saaf [25, 27]. It has been modified by Steven Bryant and Joe Eaton.

The development of this code was supported in part by the U.S. Department of Energy (DOE) through the Partnership in Computational Science (PICS) Grand Challenge Program administered through the Center for Computational Sciences (CCS) at the Oak Ridge National Laboratory (ORNL), other grants from the DOE, the U.S. National Science Foundation, and the State of Texas Governor's Energy Office.

Bibliography

[1] T. Arbogast, S. Bryant, C. Dawson, F. Saaf, C. Wang, and M. Wheeler, *Computational methods for multiphase flow and reactive transport problems arising in subsurface contaminant remediation*, J. Computational Appl. Math., 74 (1996), pp. 19–32.

[2] T. Arbogast, A. Chilakapati, and M. F. Wheeler, *A characteristic-mixed method for contaminant transport and miscible displacement*, in Computational Methods in Water Resources IX, Vol. 1: Numerical Methods in Water Resources, T. F. Russell et al., eds., Southampton, U.K., 1992, Computational Mechanics Publications, pp. 77–84.

[3] T. Arbogast, C. N. Dawson, P. T. Keenan, M. F. Wheeler, and I. Yotov, *The application of mixed methods to subsurface simulation*, in Modeling and Computation in Environmental Sciences, R. Helmig et al., eds., vol. 59 of Notes on Numerical Fluid Mechanics, Braunschweig, 1997, Vieweg Publ., pp. 1–13.

[4] ——, *Enhanced cell-centered finite differences for elliptic equations on general geometry*, SIAM J. Sci. Comput., 18 (1997). To appear.

[5] T. Arbogast, C. N. Dawson, D. Moore, F. Saaf, C. S. Soucie, M. F. Wheeler, and I. Yotov, *Validation of the pics transport code*, tech. rep., Department of Computational and Applied Mathematics, Rice University, Houston, Texas, 1993.

[6] T. Arbogast, C. N. Dawson, and M. F. Wheeler, *A parallel multiphase numerical model for subsurface contaminant transport with biodegradation kinetics*, in Computational Methods in Water Resources X, Vol. 2, A. Peters et al., eds., Dordrecht, The Netherlands, 1994, Kluwer Academic Publishers, pp. 1499–1506.

[7] ——, *A parallel algorithm for two phase multicomponent contaminant transport*, Applications of Math., 40 (1995), pp. 163–174.

[8] T. Arbogast, P. T. Keenan, M. F. Wheeler, and I. Yotov, *Logically rectangular mixed methods for darcy flow on general geometry*, in Proceedings of the 13th SPE Symposium on Reservoir Simulation held in San Antonio, Texas, February 12–15, 1995, pp. 51–59. SPE 29099.

[9] T. Arbogast and M. F. Wheeler, *A parallel numerical model for subsurface contaminant transport with biodegradation kinetics*, in The Mathematics of Finite Elements and Applications VIII (MAFELAP 1993), J. R. Whiteman, ed., New York, 1994, Wiley, pp. 199–213.

[10] ——, *A characteristics-mixed finite element method for advection dominated transport problems*, SIAM J. Numer. Anal., 32 (1995), pp. 404–424.

[11] T. Arbogast, M. F. Wheeler, and I. Yotov, *Logically rectangular mixed methods for groundwater flow and transport on general geometry*, in Computational Methods in Water Resources X, Vol. 1, A. Peters et al., eds., Dordrecht, The Netherlands, 1994, Kluwer Academic Publishers, pp. 149–156.

[12] ——, *Logically rectangular mixed methods for flow in irregular, heterogeneous domains*, in Computational Methods in Water Resources XI, A. A. Aldama et al., eds., vol. 1, Southampton, 1996, Computational Mechanics Publications, pp. 621–628.

[13] ——, *Mixed finite elements for elliptic problems with tensor coefficients as cell-centered finite differences*, SIAM J. Numer. Anal., 34 (1997), pp. 828–852.

[14] L. C. Cowsar, C. A. S. Soucie, and I. Yotov, *Parcel v1.04 user guide*, Tech. Rep. TICAM 96–28, Texas Institute for Computational and Applied Mathematics, The University of Texas at Austin, Austin, Texas, June 1996.

[15] C. Dawson, *Godunov-mixed methods for advection-diffusion equations in multidimensions*, SIAM. J. Numer. Anal., 30 (1993), pp. 1315–1332.

[16] C. Dawson and M. F. Wheeler, *An operator-splitting method for advection-diffusion-reaction problems*, in MAFELAP Proceedings VI, J. A. Whiteman, ed., Academic Press, 1988, pp. 463–482.

[17] ——, *Time-splitting methods for advection-diffusion-reaction equations arising in contaminant transport*, in Proceedings, ICIAM '91, R. O'Malley, ed., SIAM, 1992, pp. 71–82.

[18] J. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, 1983.

[19] A. S. El-Bakry, R. A. Tapia, T. Tsuchiya, and Y. Zhang, *On the formulation and theory of the Newton interior-point method for nonlinear programming*, Journal of Optimization Theory and Applications, 89 (1996), pp. 507–541.

[20] R. Glowinski and M. F. Wheeler, *Domain decomposition and mixed finite element methods for elliptic problems*, in First International Symposium on Domain Decomposition Methods for Partial Differential Equations, R. Glowinski et al., eds., SIAM, Philadelphia, 1988, pp. 144–172.

[21] F. M. Morel and J. G. Hering, *Principles and Applications of Aquatic Chemistry*, Wiley, 1993.

[22] J. C. Parker, *Multiphase flow and transport in porous media*, Reviews of Geophysics, 27 (1989), pp. 311–328.

[23] D. W. Peaceman, *Interpretation of well-block pressures in numerical reservoir simulation*, Society of Petroleum Engineers Journal, (1978), pp. 183–194.

[24] T. F. Russell and M. F. Wheeler, *Finite element and finite difference methods for continuous flows in porous media*, in The Mathematics of Reservoir Simulation, R. E. Ewing, ed., no. 1 in Frontiers in Applied Mathematics, Society for Industrial and Applied Mathematics, Philadelphia, 1983, pp. 35–106, Chapter II.

[25] F. Saaf, *A study of reactive transport phenomena in porous media*, PhD thesis, Rice University, Houston, Texas, 1996.

[26] ——, *A user's manual for nipsf: Nonlinear interior-point solver, Fortran*, Tech. Rep. Tech. Rep. TR96–24, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, 1996.

[27] F. Saaf and S. Bryant, *A user's manual for the geochemistry module in parsim1 reactive flow and transport code*, tech. rep., Center for Subsurface Modeling, Texas Institute for Computational and Applied Mathematics, The University of Texas at Austin, Austin, Texas, Jan. 1997.

[28] W. R. Smith and R. W. Missen, *Chemical Reaction Equilibrium Analysis*, Wiley, 1982.

[29] M. F. Wheeler, T. Arbogast, S. Bryant, C. N. Dawson, F. Saaf, and C. Wang, *New computational approaches for chemically reactive transport in porous media*, in Next Generation Environmental Models and Computational Methods (NGEMCOM), G. Delic and M. Wheeler, eds., Philadelphia, 1997, Proceedings of the U.S. Environmental Protection Agency Workshop (NGEMCOM), SIAM, pp. 217–226.

[30] W. I. Zangwill, *Nonlinear Programming: A Unified Approach*, Prentice-Hall, 1969.

Index