Princeton Transport Code Tutorial

Stuart A. Stothoff

Contents

1 Introduction

The Princeton Transport Code (PTC), a three-dimensional groundwater flow and contaminant transport simulator, requires several steps before any simulation can be performed. The purpose of this document is to provide a tutorial for installing, compiling, creating data sets for, and running PTC. The tutorial assumes some familiarity with the operating system that PTC is to be run under and familiarity with a text editor. The document is a companion to the full Princeton Transport Code documentation, which discusses the code in greater detail.

PTC is written in FORTRAN 77, thus may be run successfully on a variety of computers with little or no modification. This document assumes that PTC is being used on a personal computer running DOS, for the purposes of demonstration. The details of installation and compilation are system-dependent, of course, but the same general steps will need to be performed for each operating system and each compiler.

As the code is expected to be run on a variety of systems, the source code and a set of example data sets are provided. It is the responsibility of the user to select a compiler and a text editor that meets his or her individual needs. It is impossible to run PTC without these auxilliary programs. The documentation for these programs must be consulted for any details of using the individual program.

This document is also a companion to the FEPER graphics code documentation. FEPER is a graphics package which is distributed concurrently with PTC. Although the two packages work hand-in-hand with each other, they were developed independently and can be used with other programs quite successfully. At the time of this document's creation, FEPER runs on IBM and IBM-clone personal computers, and on Silicon Graphics workstations. Future versions may extend the code to other graphics environments.

Although PTC can be compiled with a variety of compilers, FEPER must be compiled with the Lahey compiler on PCs. As this compiler has been found to work well with PTC as well, a number of short-cuts are provided for using PTC with Lahey. Of course, this is not meant to slight any other compiler.

2 PTC Installation

The distribution package for PTC includes the FORTRAN source code for PTC and a collection of data sets. The data sets are discussed in detail in a subsequent section of this document.

Before PTC can be run, it is good practice to transfer the code to the hard disk for the computer. Although it is possible to run PTC using only diskettes, it is much slower and memory considerations become more stringent. In order to transfer the code from the distribution diskettes to the hard disk of a PC running DOS, one need only create the directory for the code and copy the code to this directory. Such a sequence might be as follows:

```
C > mkdir ptc
C > copy a:*.* ptc
```

Some users may prefer to install the source code in one directory and the data sets in another.

A further step is desirable if the executable program is to be used in several directories. So that the executable program can be found in any directory, the PATH variable must be adjusted to include the new PTC directory. The AUTOEXEC.BAT file, typically found in the root directory, should be editted to change this variable. Any text editor is appropriate for this. In the AUTOEXEC.BAT file, there should be a line that looks something like `PATH=C:\OS386;C:\WINDOWS;` and so forth. At the end of the line, append `C:\PTC;` (or the directory name you have chosen).

A simple installation is provided for those users that adopt Lahey. By entering the command `a:install`, a PTC directory is created on the C: drive, subdirectories entitled OBJ, SRC, and DATA are created, and the code is installed into the new directories. There should not be an existing PTC directory on the C: drive. The user still must alter the AUTOEXEC.BAT file as described above.

3 PTC Compilation

3.1 Compilation and Filenames

Compilation converts the source code, the text files provided on the distribution diskettes, into object code. Object code is the translation of the FORTRAN code into machine-level instructions. A further step **links** the object code files into a single executable program. The result of the linking step, the executable program, is what is actually used when performing simulations. The compiler that is selected will perform both of these functions. Note that object files are only an intermediate step in creating executable files, and can be safely deleted once the executable file exists.

In general, compilers don't care much about what the files are called. A DOS convention that is generally followed is that source files have "`.FOR`" appended, object files have "`.OBJ`" appended, and executable files have "`.EXE`" appended. For example, a simple program with a base name of "foo" might have a source file name of "`foo.for`", an object file name of "`foo.obj`", and an executable file name of "`foo.exe`". To run the program, one simply enters "`foo`" at the DOS prompt.

PTC uses a non-standard extension to FORTRAN 77, the INCLUDE command, to ensure that all dimension statements and each common block is consistent throughout the program. Although this is non-standard, most if not all compilers support a form of the INCLUDE statement; however, the name of the included file is subject to disagreement. PTC has two included files, the “PTCCOM” and the “INCOMM” files. Many compilers expect these files to be named PTCCOM and INCOMM, respectively. The Lahey compiler expects that these files are named PTCCOM.FOR and INCOMM.FOR, respectively.

Detailed instructions for using a compiler should be provided in the compiler documentation.

3.2 Compilation Makefiles

A set of makefiles are provided to simplify the process of compilation. A makefile collects all of the instructions for creating an executable file into one location, so that the user need only enter “make” (for example) at the DOS prompt to perform all of the steps in compilation. These are files used with compilers available to the developers of PTC, and are not intended to be a complete set covering every compiler.

When using the Lahey compiler or extended memory Lahey compiler, the makefile.lah file provided with the distribution diskette must be copied to a file called makefile in the PTC directory. In order to create the executable file not taking advantage of extended memory, enter “make ptc” at the DOS prompt. This creates an executable file called PTC.EXE in the PTC directory.

When using the extended memory Lahey compiler, enter “make ptcem” at the DOS prompt, which creates a file called PTCEM.EMF in the PTC directory. In this latter case, PTC is run by entering “os386” once, to activate extended memory capabilities, then entering “up ptcem” to run a PTC simulation. Note that using extended memory compilation allows for considerably larger problems to be considered; however, one may find that certain memory-resident software cannot be used simultaneously with the PTC simulator (Windows, for example).

When using the Unix f77 compiler, the file called makefile.unx provided with the distribution diskette must be copied to a file called makefile in the PTC directory. In order to create the executable file, enter “make ptc” at the Unix prompt. This creates an executable file called PTC in the PTC directory.

When using some other compiler, the file called makefile.gen provided with the distribution diskette may be copied to a file called make.bat in the PTC directory. This file is a template for creating a batch file to compile and link the source code for PTC into an executable; the file must be modified to fit the needs of the particular compiler available to the user. Once the file is modified, enter “make” at the DOS prompt to create the executable file.

3.3 Compile-time Customatization of PTC

PTC is set up so that a certain amount of customization of the code is performed at compile time. For example, the internal memory may be adjusted for individual computers. The available memory is assigned to one large vector, *jlrg*, which is partitioned into problem-dependent arrays at runtime. The size of this array is specified at compile-time. The executable version of the code can then be set to the largest memory allocation that the particular computer will allow.

A number of compile-time parameters are defined near the top of the `PTCCOM` source file. These parameters control the precision of the floating point data (single- or double-precision), and the size of the large array. The relevant parameters are documented in `PTCCOM` and the PTC documentation.

A further compile-time parameter is defined at the top of the `INCOMM` source file. This parameter (*ZOPSYS*) specifies the compiler that is used, so that input/output is handled correctly for the different compilers.

A final compile-time parameter is defined in subroutine `PRESET`, near the top of the `PTC.FOR` source file. This parameter (*yttred*) determines how the input command file is presented to PTC. If *yttred* is true, PTC expects the command file to be presented in the form "`ptc < input_file_name`" at a DOS prompt. If *yttred* is false, PTC expects the command file to be called `ptc.run` and a simulation is presented to PTC in the form "`ptc`" at a DOS prompt.

In order to reset the compile-time parameters, the appropriate source file must be editted (using a text editor). In each case, the change is only a minor alteration to the code, and there are instructions for making the change at the appropriate location within the source file. Note that one must both recompile and relink the object files before the effect of the parameter change is felt in the executable code!

4 Creating PTC Input

PTC is designed so that a wide variety of input and output specifications can be used, depending on the whims of particular applications. This flexibility allows the PTC command file to be customized, depending on the individual user's needs and preferences. In general, two broad classes of input can be identified, simple input and complex input, and the approach to creating PTC input files in these two cases is quite different.

PTC input is a series of command statements, one after another, within a command file. Some commands have no associated data, some commands have one or a few associated numbers, and some commands have a large set of associated numbers. Simple input occurs when only the first two types of commands are used, and complex input occurs when the third type of command is used. Usually, simple input is created "by hand" and complex input is created from external programs. In order to cut down on editting time considerably, place the simple input within the command file and the complex input in auxilliary data files. With this strategy, creating new data for a complicated parameter field with an external program has no effect on the PTC command file.

The structure for a particular command file is almost entirely up to the user. A structure that has been found to be relatively efficient, however, breaks the input into several sections. In the first section, PTC initialization occurs, and input and output parameters are specified, along with problem dimensions. In the second section, model geometry and material properties are specified. In the third section, boundary conditions and initial conditions are specified. In the final section, simulation parameters are specified and simulations are run.

5 Creating the First Data Set

In this section, a simple data set is built from scratch. Although simple, each type of boundary condition is present, and many of the considerations involved in creating the data set are discussed. The highly stylized scenario in this data set is not representative of many natural systems. As presented, the data set might simulate the case of a dry side canyon opening into a river plain.

The problem of interest has three geologic layers, two sand layers separated by a clay aquitard. The aquitard has a window of sand, where the clay layer pinches out. There is a pumping well in the bottom layer, a river crosses the top layer, and a contaminated holding pond sits above the top layer. An aquitard lies below the lower sand layer. Infiltration occurs uniformly across the top layer. There is a contamination source on the banks of the river. It is assumed that there is no flux across the sides of the domain.

In order to create this data set, an ASCII file must be opened (using a text editor). Any word processing program can be used to create the command file, as long as the file is saved in ASCII mode. PTC expects this file to be called “ptc.run”. All of the subsequent commands must be included the data file.

5.1 PTC Initialization

A number of parameters are initialized in this section. Output files are specified, problem dimensions are specified, and a few constants are input.

There are a number of output streams from PTC. Depending on individual needs, any output stream can be directed to a file, directed to the terminal, or turned off entirely. In all cases, the particular output stream needs to get a unit number referring to the destination of the output. In order to direct an output stream to a file, the file must be opened with the unit number. In the special case of output to the terminal, no file is opened but the unit number must be 6. If the unit number is negative, the output stream is turned off.

In this simple example, we choose to send all output to a file named “ptc.out”, except for the results of mass balance calculations. Mass balance calculations for flow simulations we choose to send to a file called “ptc.mbf”, and mass balance calculations for transport simulations we choose to send to a file called “ptc.mbm”.

The following snippet of a data file performs these actions.

```
open                    *open a file with unit number of 3
 3 ’ptc.out’
cecho                   *echo commands to unit 3
3
efile                   *echo input errors to unit 3
3
open                    *open an output file for flow mass balance
11 ’ptc.mbf’
open                    *open an output file for transport mass balance
12 ’ptc.mbm’
mwrit                   *direct flow and trans. mass bal. to 11 & 12 resp.
11 12
rwrit                   *output flow and transport sol’ns to unit 3 & 3 resp.
3 3
owrit                   *output ecxxxxxx requests to unit 3
3
```

Each of the **open** commands associates a unit number with a file name, and opens the file. The other commands associate a particular output stream with a unit number.

Each command has the form of an eight-character command on one line, followed by a set of numbers and characters on the next line. The commands may be capitalized or not, as desired. A word of caution is in order – the commands must have blanks following command words shorter than 8 characters, and some text editors, notably vi, do not replace tab symbols with blanks. This lack of blanks can be mysterious to track down; what appears to be a perfectly good command can result in the message “`Command not found: bad_command`”, where `bad_command` is the offending command. After the eighth character, however, any comments are acceptable.

The first **open** command associates unit number 3 with the file “ptc.out”. It is important to put single quotes around the file name. The unit number and the file name can be separated by blanks or by commas. The second and third **open** commands similarly open files “ptc.mbf” and “ptc.mbm”.

There is an output stream which consists solely of the eight-character command names, echoed as they are read from the data file. The above **cecho** command directs this output stream to unit 3. Another common choice is to direct this stream to unit 6, the terminal, so that if there is a visual indication of the progress of the program. This can be helpful when a data set is not working as expected. For those confident that everything is running smoothly, directing the output stream to unit -3 (for example) will shut off the stream.

The **efile** command is for data errors that are caught on input, before interpretation. Relatively few errors go to this output stream.

The **mwrit** command directs the output from the mass balance calculations. Flow calculation results and transport calculation results should each get their own file, as the results are difficult to follow otherwise.

The **rwrit** command directs the output from the simulator calculations. These are the actual simulator predictions, in tabular form suitable for viewing by the human eye. If the results are to be massaged by a post-processor program, however, the alternate command, **gwrit** (discussed in a subsequent section), is more suitable.

The **owrit** command directs the output from specific requests for output. One can request that a given variable be printed out at any point in a data set, and the destination of this output is controlled by this command. This command is extremely handy when debugging a command file, as the user can interactively interrogate what information PTC believes that it has received. This procedure is discussed in the section on debugging.

There are restrictions on the available unit numbers. PTC reserves the use of unit numbers 1, 5, 15 through 19, and 91 through 99 for internal use. In addition, unit 6 is used for output to the terminal, and must not be used for data input.

Now that the decisions on most of the output streams have been made, a number of parameters can be input. We can now input the title of the problem, the dimensions of the problem, and a few constants. We are simulating a problem which will have a mesh with 12 elements in the horizontal plane, 20 nodes in the horizontal plane, and 3 layers stacked on top of each other, and we will call this simulation "Simple Problem". Note that the mesh for this problem is hopelessly coarse – this is only an example to demonstrate input!

The following bit of a data file performs these actions.

```
rdtitle                 *input title
 'Simple Problem'
setfdon                 *dimension for all flow variables
setmdon                 *dimension for all transport variables
setdfrac                *max frac of nodes in a layer which are dirichlet
 .2
rddims                  *specify number of nodes, elements, layers
 20 12 3
rddifusn                *molecular diffusion
 .0001
rdweight                *upstream weight (default = 1.)
 1.
```

In the above, the title and the dimension statements are self-explanatory. Again, the title must be surrounded by the single quotes. The quotes must be the type that slant up to the right – you may get a mysterious result if the wrong types of quotes are used!

Before the dimensions are specified, the configuration of the memory must be set up using the **setfdon**, **setmdon**, and the **setdfrac** commands. The **setfdon/setfdoff** commands specify that dimensioning is required/not required for flow simulations this command session. If a flow simulation will be run from the command file, **setfdon** is required; if no flow simulation will be run, **setfdoff** does not allocate the otherwise needed memory. **Setmdon/setmdoff** perform the corresponding functions for mass transport. The **setdfrac** command specifies the fraction of nodes in a layer which will be denoted dirichlet nodes; the worst-case layer is used. This last command is only important when mass balancing is used. By default, configuration is for both flow and transport, and up to 20% of the nodes are allowed to be dirichlet.

It is extremely important that the dimensions are input as one of the first commands. This **MUST** be specified before any of the model geometry, material properties, boundary conditions, and initial conditions are input!

The **rddifusn** command specifies the molecular diffusion coefficient used in the transport simulation. The dispersive term in the transport equation is assumed to consist of two parts, a purely diffusive term and a dispersive term. The dispersive component is assumed to be velocity-dependent, and this is discussed in the section on Material Properties. The diffusive component, on the other hand, is assumed to be independent of velocity. Typically this term is quite small compared to the dispersive term, and the influence of this term is only felt where the fluid velocity is near zero.

The **rdweight** command specifies the upstream weighting parameter used in transport simulations. This value is in the range from 0 to 1, with 1 being fully upstream weighted. The upstream weighting parameter is used to help control non-physical oscillations in the concentration field. By adding upstream weighting, the advective term in the transport equation is weighted towards the "upstream" side of each element, by incorporating an asymmetric quadratic term to the weighting functions integrating velocity. Cautious use of this parameter is recommended, as upstream weighting acts like a dispersive term. Over-use of this parameter can result in an over-dispersed field.

There are a few other parameters that might logically be initialized in this section, or the diffusion coefficient and upstream weighting might be input in the section controlling simulation. This is a matter of user preference. These other parameters are discussed in the Simulation section.

5.2 Model Geometry

The finite element mesh is created in this section. PTC uses a splitting algorithm that conceptually solves the governing equation on a set of "horizontal" slices, using a horizontal finite element mesh for each slice, and then performs a set of vertical sweeps to link the slices. In order to do this, the nodes in each horizontal slice are stacked one directly above the other. With this assumption, the horizontal (x,y) coordinates of the slices are identical for each slice. The vertical distance between slices may vary between nodes, and may vary for each slice at a given horizontal location.

Accordingly, the model geometry is fully specified with a single horizontal finite element mesh and the elevation of the slices. This information is specified by defining the horizontal location of a set of nodes, the incidence or node list for each horizontal element, and the vertical elevation of the interfaces between layers. These interfaces include the bottom of the lowest layer and the top of the highest layer, so that one more set of interface elevations is needed than there are layers. For example, if there are 3 layers, then 4 interface elevation sets must be defined.

In our simple problem, we will put in a regular mesh for the horizontal elements and assume that the layers are all of constant thickness. This is shown in Figure 1. The bottom layer is assumed thicker than the top two layers. PTC will generate a rectangular horizontal mesh, and we will use this capability to define the model geometry. The domain of interest is of length 600 in the x direction and of length 400 in the y direction, and we will use 4 elements in the x direction and 3 in the y direction.

The following portion of a data file performs these actions.

```
*
* model geometry
*
gnincid                 *generate incidence list
 4 3                    *4 elements in x, 3 in y
gnreccor                *generate rectangular coordinates
 0. 0. 600. 400.        *corners for generation: (x1,y1), (x2,y2)
draelev                 *interface 1 elevation
 1 0.
draelev                 *interface 2 elevation
 2 70.
draelev                 *interface 3 elevation
 3 79.55
draelev                 *interface 4 elevation
 4 95.
```

The **gnincid** command creates an incidence list for a *topologically* rectangular mesh. All elements have 4 nodes. This command is sometimes of use when using a distorted mesh.

The **gnreccor** command creates a set of nodes laid out on a regular rectangular grid. The grid is aligned with the x and the y axes. The **gnincid** command must precede this command, so that the number of elements in each direction is known. Each element is exactly the same size and shape. Node numbering proceeds along the x axis first, starting at (*x1*, *y1*); however, the equation numbering is transparently defined internally, yielding the most efficient computational scheme. Elements are numbered in the same way.

The **draelev** command is used to specify the elevation of the interfaces. In this example, the first layer is 70 thick, the second is 9.55, and the third is 15.45 thick.

The **draelev** command is the first example of a distributed parameter command. Most of the parameters in PTC are defined either for each node or for each element. Material

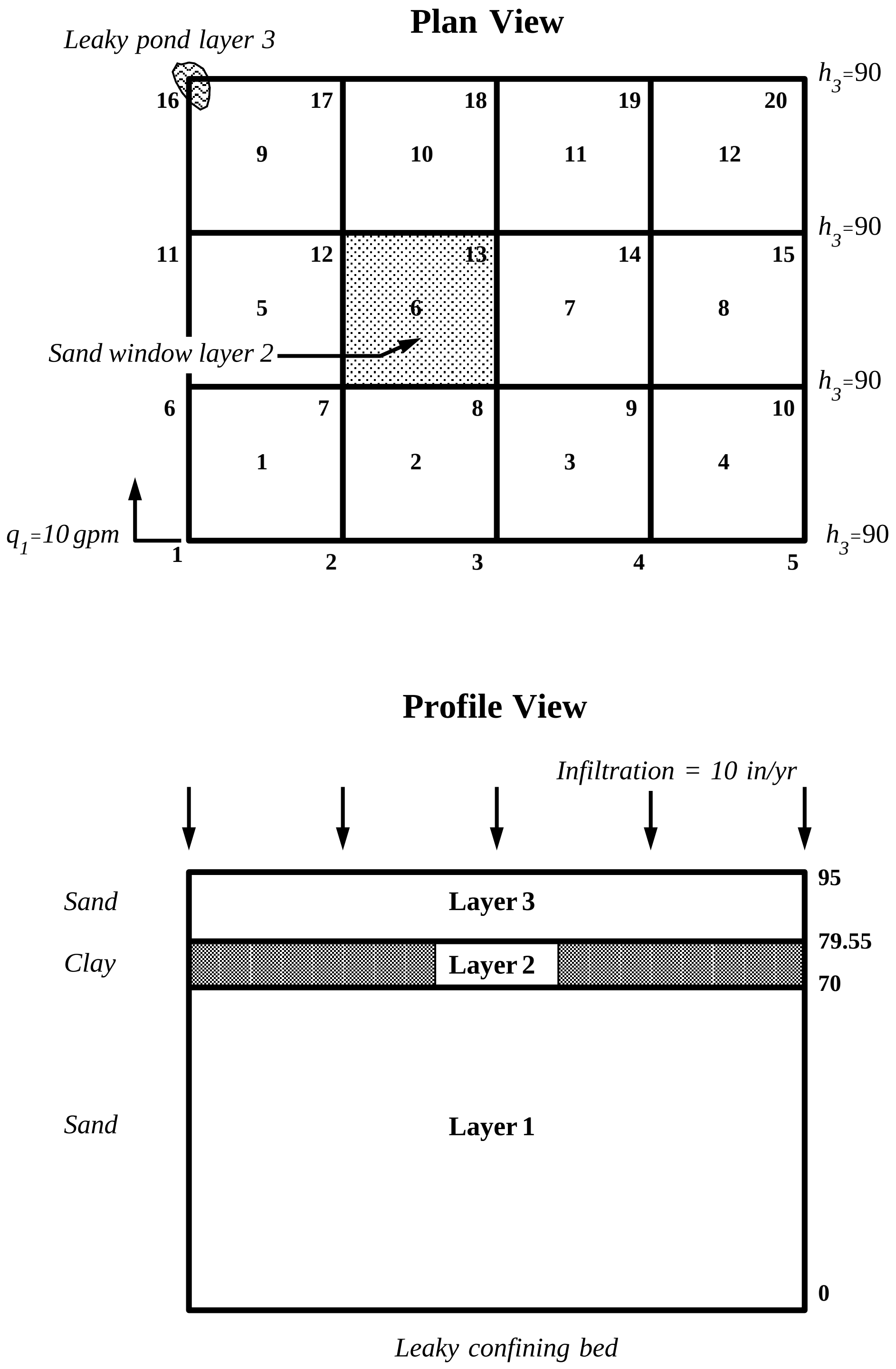

Figure 1: Mesh and boundary conditions for the simple problem.

parameters (conductivity, storage coefficient, porosity, and dispersivity) are defined as piecewise constant over elements. Model geometry parameters (nodal coordinates, elevations) are defined by nodes. Dependent variables (head, concentration) are defined by nodes as well. Velocities, which are proportional to the gradient of head, are defined as piecewise constant over elements.

All of these parameters may be input in five different formats, depending on preference. Each of these commands is broken into three parts, which specify what parameter is being input, what portion of the model has the parameter assigned to it, and whether or not to clear the array before reading in the parameter.

In this example, the **elev** portion of the command specifies that elevations are being input. The **dr** portion of the command specifies that the array for the particular interface is to be cleared before reading in the new value. The **a** portion of the command specifies that all members of the specified interface will receive the same value, and implies that an interface number and an elevation will be input on the next line. As the interfaces and layers are numbered from the bottom up, the lowest elevation is applied to the first interface. Other distributed parameters and command formats will be discussed in the next section.

5.3 Material Properties

There are a number of material properties that are defined as constant over each element. In the flow equation, distributed parameters include hydraulic conductivity and the storage coefficient. In the transport equation, distributed parameters include porosity and the longitudinal and transverse dispersion coefficients. In some cases, velocities may be considered as a distributed parameter as well. All of these parameters use the same distributed parameter protocols presented for inputting elevations.

The domain of interest is assumed to be two sand aquifers, separated by a clay aquitard. The clay aquitard has a sand window in the middle of the domain. In order to specify the distributed parameters, a series of steps are performed. First, a global value for each parameter is assigned to each element in each layer, corresponding to the sand aquifers. Then the parameters for the clay layer overwrite the original values. Finally the parameters for the window in the clay layer overwrite the values for the clay. This process illustrates some of the power of the PTC input structure.

The following portion of a data set performs these actions.

```
*
* apply fmxz, fmyz, fmzz, stz, porz, elz, etz, evz
*
drgparam            *entire model
 10. 10. 1. .001 .2 50. 10. 1.
deaparam            *layer 2
 2 .001 .001 .0001 1.e-4 .4 50. 10. 1.
```

```
*
* modify for the window (element 6) in the clay
*
descondx            *window x conductivity
 2 10.
 6/
descondy            *window y conductivity
 2 10.
 6/
descondz            *window z conductivity
 2 1.
 6/
desporos            *window porosity
 2 .2
 6/
```

The lines starting with a * are comment lines. These can be placed wherever a command is expected. One must be cautious about placing the comments, so that a comment is not inadvertently placed where data is to be read. Comments can also be placed following all of the data on a particular line.

The **drgparam** command specifies all of the flow and transport parameters for the entire model in one fell swoop. Again, the **dr** portion of the command clears all of the arrays before reading in any data (which is not necessary in this case). The **g** portion of the command specifies that the subsequent values will be applied to every element in every layer. The **param** portion of the command specifies what parameter is input (all of them, in this case). The order of parameters in this command is x direction conductivity, y direction conductivity, z direction conductivity, the storage coefficient, porosity, longitudinal dispersivity, horizontal transverse dispersivity, and vertical transverse dispersivity.

The **deaparam** command specifies all of the flow and transport parameters for the second layer at once. The **de** portion of the command indicates that the arrays are going to be editted, not cleared, so the arrays are merely overwritten where necessary. And again, the **a** portion of the command requires a layer pointer.

The **descondx** command assigns a x conductivity value to specific elements in a layer, where the **s** portion of the command is the new flag to be aware of. This implies that the layer index is needed, as well as the value to be assigned, on the next line. On the second line, the elements receiving this value are listed, with the list ending with a "/". In this particular example, only one element is listed. In general, any number of elements can be listed, and the input can extend over several lines in the data file. Any comments for the list of elements MUST follow the final "/"!

The **descondy**, **descondz**, and **desporos** follow the same protocol for y conductivity, z conductivity, and porosity respectively. Note that a common error is to try to overwrite a small portion of the data array using a **drscondx** command instead of a **descondx** command (for example). All of the previously existing values are cleared, hardly the effect one usually anticipates!

5.4 Initial Conditions

Initial conditions use the same distributed parameter protocols as do material properties, except that each of these conditions is assigned by node rather than by element.

Initial conditions are usually simple to apply. In most situations, it is sufficient to start with a clean domain for any contaminant. Often, too, a uniform initial head distribution is acceptable, since the steady-state distribution of head is what is desired. Unfortunately, PTC does not solve the steady-state flow equation directly, due to the splitting algorithm used, so time stepping is required to proceed to steady state. This point is discussed in greater detail in the simulation section.

A staged imposition of initial conditions often works well when a uniform distribution of head is not acceptable. A case where a staged approach is highly desirable occurs when a pumping test is to be simulated, and a steady-state head distribution is needed. Another case occurs when a water table simulation is being used and the water table is near the top or bottom of the top layer, and it is critical to keep the water table from moving too rapidly due to extreme disequilibrium.

In the staged approach, the uniform initial condition is used to run for a number of time steps, until steady state is achieved, using confined conditions. If the water table initial condition is needed, the water table option is turned on and the simulator is rerun with the new head distribution. The true simulation can then be run with this generated initial condition. This generated initial condition can also be saved, if desired, and used as the initial condition for subsequent simulations.

In the following data file segment, however, the simple conditions are imposed.

```
*
* initial conditions
*
drginith           *initial head for entire model
 90.
drginitc           *initial concentration for entire model
 0.
```

Both the **drginith** command and the **drginitc** command follow the distributed parameter protocols presented previously, for initial head and initial concentration respectively.

5.5 Boundary Conditions

Boundary conditions use the same distributed parameter protocols as do initial conditions. Boundary conditions and forcing functions are a little more problematic to impose than initial conditions.

The default condition for the flow simulator is that there is a no-flux boundary condition over the entire boundary, and no imposed fluxes anywhere within the domain. In the flow simulator, one may impose specified head values at any node, a flux condition at any node, or a leakage condition at any node. A special case exists for the top layer, where a specified infiltration rate may also be applied.

In the transport simulator, the default condition is no dispersive flux across the boundaries, and no applied fluxes occur anywhere within the domain. One may impose specified concentration values at any node, or the concentration of the external water source in the case of flux and leakage nodes in the flow equation. It is assumed that the infiltration water is clean.

Dirichlet, or specified value, conditions are necessary for the steady state flow equation to be well posed. The more Dirichlet conditions that are specified, the better behaved the solution will be and the closer one can make model predictions match field predictions. On the other hand, the fewer Dirichlet conditions, the more sensitive the model is to the material parameters and the more confidence one may have in predictions of flow velocities. There is a tradeoff between these two considerations that it is up to the analyst to determine when assigning boundary conditions.

A number of common hydrologic features can be accommodated with PTC. In the case of a river, any of the three boundary condition types can be used. For example, if the analyst feels that there is direct communication between the river and the aquifer, a Dirichlet condition is appropriate. If the river is well above the aquifer, so that there is no direct communication, the river may be serving as a source of recharge and a flux condition is appropriate. If the bottom of the river has a retarding layer, such as a bed of peat or muck, but there is communication between the river and the aquifer, a leakage condition is appropriate. The leakage condition requires a reference head and a conductance value, which is the conductivity of the retarding layer divided by the thickness of this layer. This condition is like a weak Dirichlet condition, in effect. These same considerations apply for other surface features, such as ponds, lakes, and drains.

In certain cases, it may be desirable to "extend" the boundaries of the domain artificially by applying a leakage condition corresponding to the aquifer between the boundary and some water source. A similar strategy may occur if the bottom of the simulation domain is an aquitard rather than an impermeable layer.

Groundwater divides are handled by aligning the edge of the mesh with the divide and letting the default boundary conditions take effect. In cases where a streamline is far from the region of interest, this strategy may be cautiously applied as well. These are both considered no-flow boundaries.

If a boundary is to have some specified gradient, however, an indirect procedure is required. In this case, the equivalent flux across the boundary must be calculated by the analyst, and the resulting flux must be distributed to imaginary wells located at the nodes. A successful strategy is to assign half the flux across an element boundary face to the two nodes forming the face.

A simple procedure for assigning infiltration to the top layer is provided. The analyst provides the infiltration rate, such as in/yr, to PTC and internally the infiltration rate is multiplied by the area around the node where the infiltration is applied. This volumetric rate is then applied internally in the same way that well fluxes are applied.

In our example problem, there is a pumping well in the bottom layer, a river crosses the top layer, and a contaminated holding pond sits on the top layer. An aquitard lies below the lower sand layer, and infiltration occurs uniformly across the top layer. In this case, the river is assumed to be in direct hydraulic connection with the top layer aquifer, so a Dirichlet condition is imposed. The holding pond has a sludgy layer at the bottom of the pond which weakens the hydraulic connection, so a leakage condition is imposed. The aquitard is in communication with another aquifer, which is presumed to be at a constant reference head, so again a leakage condition is imposed.

The portion of the command file which imposes these conditions is as follows.

```
*
* forcing functions
*
indexon
drnbcflo            *layer 3 Dirichlet bcs (river)
 3
  5  1 90.
 10  1 90.
 15  1 90.
 20  1 90.
 0/
scale               *convert gpm to ft**3/day
192.5134
denbcflo            *layer 1 flux (pumping well) for flow
 1
 1   2 -10.
0/
indexoff
scale               *convert in/year to ft/day
 2.283105e-4
drgrain             *10 in/year infiltration over top layer
 10.
```

```
scaleoff
dealeak            *aquitard below layer 1
 1  90. 1.e-3 -1.  *ref head, conductance, ref conc
desleak            *holding pond in layer 3
 3  96. .0001 20.  *ref head, conductance, ref conc
 16/
```

More of the distributed parameter commands are introduced in this snippet of the data file. The **indexon** command informs PTC that the data in the subsequent **drnbcflo** command will be indexed by node number. This condition will exist for any **drnxxxxx**, **denxxxxx**, **drexxxxx**, or **deexxxxx** commands encountered until a matching **indexoff** command is input, where the **xxxxx** can be any of the distributed parameters. Examples of the **drnxxxxx** and **denxxxxx** commands are the **drnbcflo** and **denbcflo** commands discussed below. The only difference between **drnxxxxx** commands and **drexxxxx** commands are that input is expected by node in the first case, and by element in the second.

The **drnbcflo** command imposes Dirichlet and flux boundary conditions for the flow simulator. The **n** portion of the command states that the data that is input is specified by node. As is usual with the distributed parameter commands, the layer index is specified on the line after the command. On the next lines, a list of the node number, the boundary condition flag, and the boundary condition value are specified for each node. The boundary condition flag is either 1 or 2, denoting Dirichlet or flux conditions respectively. In this case, we have specified that nodes 5, 10, 15, and 20 are to receive Dirichlet conditions. Each specified head for the river nodes turns out to be 90 (normally this might vary along a river, of course). In order to signal that input of layer 3 flow boundary conditions is over, a zero node number is specified, followed by a "/".

If the **indexon** command were not in effect, every node in the third layer would need to be specified, in order from 1 to 20, without the node index. PTC would return to command mode immediately after the 20th datum is read.

The next command, **scale**, is a handy way to convert data from one unit system to the self-consistent units used in the simulation. In this case, the field measurements are made in gallons per minute, but the units used for the simulation are feet and days. Scaling takes place on every real parameter read. Note that it is probably *not* a good idea to use scaling when the leakage parameters are being specified, as all three values are scaled!

Scaling continues for every command until turned off with the **scaleoff** command. It is important to turn off scaling *immediately* after it is no longer needed!

The **denbcflo** command is used to input the pumping well in the bottom layer. This is similar to the river specification, except that the boundary condition flag now signals flux conditions. The negative flux signals that water is being *withdrawn* from the system. It is possible to mix Dirichlet and flux conditions in the same specification; however, it may be a poor idea, especially if there is a chance that scaling will be used.

The **drgrain** command inputs the infiltration reaching the top of the domain. The command name is a slight misnomer, as one expects that there is some runoff that does not reach the aquifer. Note that a layer flag is never required for infiltration, as this is only applied at the top layer.

The **dealeak** and the **desleak** commands use the standard distributed parameter formats. In this case, three pieces of information are input: the reference head on the side of the leakage barrier not in the domain, the conductance (conductivity divided by distance) of the barrier, and the concentration of the fluid behind the leakage barrier. If the concentration is less than or equal to zero, fluid entering the system is assumed clean. Fluid leaving the system always has the concentration of the fluid within the system. In this example, the aquifer below the aquitard is clean, but the holding pond is somewhat contaminated. The leakage conditions are the only case where both flow and transport boundary conditions are specified simultaneously; if the transport simulator is not used for a particular simulation, the reference concentration is arbitrary.

The possible boundary conditions for the transport simulator are simpler than for the flow simulator. Either the concentration is specified at a node, or the concentration of incoming water is specified. The default boundary condition is that dispersive flux of the contaminant is zero, although the contaminant is advected past the boundary if a non-zero flux of water is present. The concentration of exitting water is assumed to be that of the ambient concentration field.

For hydrologic sources, such as holding ponds, usually the concentration of the source water is specified. In cases where there is no convenient hydrologic source, such as surface pits or spill sites, there are basically two approaches to introducing the contaminant. If the mass of the contaminant entering the system can be reliably estimated, the appropriate boundary condition is achieved using an imaginary well. A very small influx at the site of the contaminant is introduced, with a high concentration, thereby introducing the correct amount of mass. If the amount of mass entering the system is unknown, or if the source of contamination is so strong that the dissolution limit of the contaminant is approached, a Dirichlet condition is more appropriate.

In some cases, several adjacent nodes may be used to spread the source around. This can ease numerical problems associated with the sharp gradients that often occur at early time.

Experience shows that it is vital to tack down the concentration field in a larger simulation at several places, particularly on the upstream portion of the domain. This can be conveniently done by specifying zero concentration Dirichlet nodes far from the region where contaminant is expected.

In our example, the contaminated holding pond has already been accounted for in the leakage boundary condition. The pumping well needs no transport boundary condition. The only remaining boundary condition is that of the source at the banks of the river, as the simulation is not large enough to require extra zero concentration nodes. This is specified as follows.

```
drsbccon            *layer 3 specified concentration
```

```
 3   1 100.
 15/
```

As in the **bcflo** commands, the **bccon** command has a boundary condition flag and an associated value. In this case, the boundary condition flag is 1, signifying that a specified concentration, or Dirichlet, value is to be applied at a node. If the boundary condition flag were 2, this concentration value would be the concentration of the incoming water at that node. Note that in the latter case, there is no effect for nodes without an associated flux condition.

5.6 Specifying Output

The PTC session presented thus far is only of moderate use, despite creating a potentially interesting set of simulations. The only information that PTC has output are the results of the mass balance calculations. It is assumed that nothing at all will be output unless PTC is specifically requested for the information.

Any input variable can be echoed if desired. In addition, head, velocity, and concentration can all be output at regular intervals during an individual simulation. Any of the distributed variables can also be output in a form suitable for graphical post-processing, either as nodal values or as elemental values (appropriate averaging is done, if required, to convert elemental values to nodal values or vice versa). The user defines the form of this output. This procedure is discussed in the complex example.

At the start of our simple example problem, PTC was told where output requests should be directed (file “ptc.out”). In order to get an echo of what PTC feels has been input, a set of commands can be used as follows.

```
*
* output commands
*
eccoors             *echo nodal coordinates
ecincid             *echo incidence list
ecelev              *echo interface elevations
eclayer             *echo material properties at all elements
ecbc                *echo boundary conditions
ecrain              *echo infiltration flux
ecinith             *echo initial head at all nodes
ecinitc             *echo initial concentration at all nodes
```

Most of these commands have direct counterparts with the input commands. The **ecelev** command is the counterpart to a **draelev** command, the **ecrain** command is the counterpart to a **drgrain** command, and so on.

The **ecbc** command outputs all flow and transport boundary conditions, combining all of the boundary condition commands. The **eclayer** command is the counterpart of the **drgparam** command, outputting all material parameters at once. Each of the material properties may be examined individually, as well. A full list of the commands for echoing input data is provided in the PTC documentation.

An important consequence of the command-driven style of input to PTC becomes evident when requesting output, as the same output can be requested repeatedly in order to debug the input set. This point is discussed more fully in the section on debugging. However, this same flexibility means that there is no specific place in the data set to request output. Perhaps the best place to request documentation of the input data is immediately before simulation, so that all possible problems with the input have had a chance to manifest themselves.

As stated before, the head, velocity, and concentration fields can be output at regular intervals during simulation. This is done in a two-part procedure. First, the output units must be set (as shown in the PTC Initialization section). Secondly, the initial step for output and output interval must be specified. In our case, we only want output every fifty steps, for all three fields. The following bit of a command file does this.

```
rdprtkey            *specification of solution output intervals
 50 50 50 50        *kftbeg kftinc kmtbeg kmtinc
rdvopkey            *specification of velocity output intervals
 50 50              *kvtbeg kvtinc
```

The **rdprtkey** command specifies the output interval for both flow and transport simulations. Of course, if only one of the two is being run, the parameters for the other are arbitrary. The *kftbeg* parameter is the first flow time step where head solutions are to be output, and the *kftinc* parameter is the interval between outputs after the first. The *kmtbeg* and *kmtinc* variables are the corresponding parameters governing concentration output, except that the *transport* time steps are used. The **rdvopkey** command is exactly analogous; again the *flow* time step is used.

Sometimes the mass entering the system at Dirichlet nodes is of interest, such as when the quantity of recharge from a stream is to be estimated. The following commands output these quantities, as long as the mass balance option has been enabled.

```
*rdqopkey           *calculated flux at flow Dirichlet nodes
*1 1                *kqtbeg kqtinc
*rddopkey           *calculated dispersive flux at trans. Dir. nodes
*1 1                *kdtbeg kdtinc
```

Both **rdqopkey** and **rddopkey** are exactly analogous to the **rdvopkey** command. Output from these commands goes into the general output file, as specified by the **owrit** command. These are commented out in this example.

The output commands can be applied wherever desired. A logical place to locate output commands is amidst the simulation control commands, so that all variables have been specified and all output commands are in one place.

5.7 Simulation

Once the model geometry, material properties, initial conditions, and boundary conditions have been specified, actually running a simulation requires little more information. PTC needs to know exactly which simulation options are to be used, and needs to know what the time stepping strategy will be for the simulation.

The strategy that will be followed in our example will be to first run a flow simulation to approximate steady state, then run a transport simulation with these steady state conditions. We will assume that the water table option is not used.

The final commands required to perform a flow simulation for our simple problem are as follows.

```
*
* flow simulation
*
*deastor             *layer 3 (water table storage coefficient)
* 3 .2
*rdwtable
* 0 50   .001        *ytable niter epsiln (default is wtable off)
rdsteady
 .001               *dhdtss
rdcontrl
 1 0 0 1 1          *yflow yvel yconc yincor ymbal
gntime
 200 10 100 2 1.5 36500 *itmax itchng itchmx nmspf chng tleng
sim
```

The **rdcontrl** command specifies which simulator calculations will be performed. The first three options specify whether flow calculations, velocity calculations, and transport calculations will be performed. In this **rdcontrl** command, we are only running the flow simulator.

The fourth option specifies that calculations are to be performed entirely within RAM; in extreme cases, where available memory is limited, it is possible to store certain large arrays on the hard disk. PTC has a flag in the PTCCOM module which eliminates most of the required storage by turning off the flag and recompiling. As accessing disk storage memory is considerably slower than accessing RAM memory, it is highly recommended that all calculations be performed in core whenever possible.

The final option specifies whether mass balance calculations will be performed. Mass balance output occurs each time step. It is usually a good idea to have mass balancing done, as there is not a large performance cost associated. Memory considerations are perhaps more important in deciding whether mass balancing is necessary. A number of additional arrays are required for mass balance calculations, so the version of PTC must be compiled with the mass balance flag in PTCCOM set to enable mass balancing.

The **rdsteady** command provides a rate of change of head with respect to head parameter. If the absolute value of the largest rate of change of head anywhere within the domain over a time step is less than this value, the flow calculations are considered unnecessary. The head field remains unchanged for the rest of the time steps requested for the simulation, and for any subsequent simulations. Mass balance calculations continue, with the current head field. If any boundary condition, material parameter, or flow initial condition is changed following the current simulation, the steady state condition is released. The current head field can be forced to be considered steady state by using the **setsson** command; this is released by using the **setssoff** command.

The **rdwtable** command is used when unconfined conditions are present in the top layer. The flags associated with the water table command turn on or off the use of the water table option, determine the maximum number of nonlinear iterations allowed for the water table to converge within a time step, and the criterion which specifies convergence. In this case, the water table is explicitly turned off, as the first flag is zero, so the other parameters are arbitrary. If the first flag were one, PTC could use up to 50 iterations to have the maximum water table elevation change over an iteration be less than 0.001. Usually only 2 or 3 iterations are required.

The commented-out **deastor** command is associated with the **rdwtable** command. When the water table option is on, the storage coefficient represents the effective porosity, *not* the compressibility of the water and porous medium. This is due to the storage effect obtained from occupying additional pores when physically raising the water table. Since porosity is orders of magnitude larger than compressibility effects, one finds that the response of the system is orders of magnitude slower in an unconfined simulation than in the equivalent confined simulation. It is important to remember to include the correct storage coefficient in transient simulations!

One further command, not shown in this data set, is of use when part of the domain is confined and part is unconfined. Nodes can be individually designated either water table or confined, using the **rdwtnode** command. An array of ones and zeroes is expected, using the drnxxxxx protocols for distributed parameter input. If a particular node is assigned a 1, the node is considered unconfined.

Time stepping in PTC trades off rapidly responding flow field behavior with the relatively slower concentration field response. Every time a time step changes, there is a computational burden associated, as the coefficient matrix must be reassembled and decomposed again, with the computational burden for transport roughly four times that for flow. If the time step remains constant, calculating the new head or concentration field is relatively inexpensive. A strategy to minimize the burden only changes the time step size every few time steps, squeezing some benefit from the relatively inexpensive portion of the updating process. The variable time stepping is felt to be more important in the flow simulator, as the flow field tends to have an exponentially decaying response to perturbations, such as might occur by turning on a well. Since usually the flow time step is considerably larger than the transport time step, time stepping in PTC is based on the flow time step size. The transport time step is defined by the number of transport time steps occurring within one flow time step: if the time step sizes are the same, there is one transport step for each flow step. If there are more, it is assumed that they are each of the same size.

The **gntime** command specifies the time-stepping strategy for the flow simulation. The *itmax* parameter is the total number of time steps for the simulation. The *tleng* parameter is the total length of time that the simulation is to proceed over. The *nmspf* parameter is the number of transport time steps per flow step, which is arbitrary in this case as the transport simulator is not being used. The *itchng*, *itchmx*, and *chng* parameters control the increase in time step alluded to above. The *itchng* parameter determines the number of constant time steps between changing the time step size. The *chng* parameter is the multiple of the previous flow time step size used when increasing the time step; a common value is 1.5. The *itchmx* parameter specifies the number of time steps for which this increasing time step strategy will occur over. So, in this example, each 10 time steps the current time step size will be multiplied by 1.5; between 50 time steps and the end of the simulation, the time step size will remain unchanged. There will be 100 total time steps, and the initial time step is chosen internally so that the total length of simulation will come out to 10 years.

The **sim** command tells PTC to run the simulation with the current settings on all parameters.

Following the flow simulation, we will run a transport simulation. The appropriate commands are as follows.

```
*
* transport simulation
*
rdcontrl
 1 1 1 1 1          *yflow yvel yconc yincor ymbal
rdtime
 1 5 50 100 365 1.5 *itmax itchng itchmx nmspf delt chng
sim
end
```

In this **rdcontrl** command, all options are on. In order to run transport simulations, the velocity field must be available. In our case, the velocities are computed based on the head distribution, and this calculation must be explicitly specified. If the velocity field comes from a data set, the flow and velocity calculations are not needed. In fact, the flow simulation is really only needed to initialize certain mass balance quantities.

In the **rdtime** command, the initial time step size for flow is explicitly specified (*delt*), rather than the total length of the flow simulation period (*tleng*). The other variables are identical to the **gntime** command parameters. In this particular case, we are taking one flow step, of length 356 days, with 100 transport steps (each of length 3.65 days). The other variables are arbitrary, as they only come into play when there are multiple flow steps.

In the second **sim** command, the flow field resulting from the first simulation is used as "initial conditions" for the single new flow step. If steady state has been reached, according to the **rdsteady** command criterion or the **setsson** command, the flow field is not actually solved for. Experience shows that a large change in the head field can occur when starting the transport simulation, unless PTC takes advantage of the approximate steady state condition. As the time step is much larger, one year as opposed to several days, the same rate of change of head will produce changes much larger than previously observed. In addition, the effect of the time derivative in the flow matrix is much smaller with the large time step, and the splitting algorithm becomes less stable with the reduced time effect. For these reasons, it is important to achieve the steady state criterion when performing the flow simulation.

The **end** command completes the PTC session and ends the program. Synonyms are **quit** and **stop**.

5.8 Running the Command File

The first data set has been completely specified. Assuming that PTC has been compiled, and the executable version is either in the current directory or can be found through the path variable, this data set is invoked simply by entering `ptc` (or whatever the executable code is called) at the DOS prompt.

In order to test that PTC is running correctly, a sample command file called "ptc-simp.run" is provided. This is a copy of the command file that has been built in this tutorial. If there is some difficulty in creating the sample command file from scratch, this command file can be consulted.

When several data sets are to be used, either the data sets can be stored in different directories or the command files can be saved with different names, being copied into the "ptc.run" file as needed. In order to run the provided sample command file, simply copy it to "ptc.run". Make sure that whatever file already called "ptc.run" is saved before overwriting it.

6 Creating a Complex Data Set

The second example problem builds upon the concepts presented in the simple example, but many of the distributed data parameters are input in a way that is more suitable for larger, more complex models. The problem presented in the simple example is reworked, with higher resolution and a curvilinear mesh. Each of the geologic layers is split in two for this example. Essentially the same physical scenario is present; however, periodic forcing functions are imposed, thus incorporating seasonal variation. This example emphasizes some of the power of the PTC input structure.

6.1 PTC Initialization

The initialization of this more complex data set follows the same procedure as in the simple data set. Again, the output files are opened, output streams are assigned, and a few parameters are initialized. The command file is as follows.

```
open                    *open a file with unit number of 3
 3 'ptc.out'
cecho                   *echo commands to unit 3
3
efile                   *echo input errors to unit 3
3
open                    *open an output file for flow mass balance
11 'ptc.mbf'
open                    *open an output file for transport mass balance
12 'ptc.mbm'
mwrit                   *direct flow and trans. mass bal. to 11 & 12 resp.
11 12
rwrit                   *output flow and mass solutions to unit 3 & 3 resp.
3 3
owrit                   *output ecxxxxxx requests to unit 3
3
setfdon                 *dimension for all flow variables
setmdon                 *dimension for all transport variables
setdfrac                *max frac of nodes in a layer which are dirichlet
 .2
rdtitle                 *input title
 'Complex Problem'
rddims                  *specify number of nodes, elements, layers
 91 72 6
rddifusn                *molecular diffusion
 .0001
rdweight                *upstream weight (default = 1.)
 1.
```

Only the title and dimensions have changed from the simple problem.

6.2 Model Geometry

The model problem uses a curvilinear mesh rather than the rigid grid used in the simple example. There is a slight concentration of elements in the vicinity of the clay window. In this case, outside computer software has created the model geometry, including the incidence list, the nodal coordinates, the layer elevations. An optional node renumbering list might also be created with external software, in order to optimize the bandwidth of the coefficient arrays.

For ease of modification, we choose to keep each of these variables in separate files, external to the command file. If one of these variables is changed, a new file incorporating the changes can simply overwrite the current file, without entering a text editor.

The three "field" layers, two sand layers and a clay aquitard, are each split into two computational layers. In the same way that using smaller and smaller elements provides a more accurate representation of the horizontal head and concentration fields, using more layers provides a more accurate vertical representation. Due to the splitting algorithm that PTC uses, doubling the number of layers doubles the computational effort. The most straightforward strategy is to input the parameters based on the field layers, then split the layers afterwards. With this strategy, layers 1, 2, and 3 are the field layers: all material properties, initial conditions, and boundary conditions are specified on layers 1, 2, and 3. After these values are specified, the layers are split. Finally, any values specific to a particular computational layer are adjusted.

In order to input these variables, the command file might look like the following.

```
open                  *open the file with incidences and coordinates
 9 'mesh.dat'
dfile                 *specify the data file and format
 9 '*' 1              *munit zfmt ngpl
indexon
rdincid               *read incidences
rvincid               *reverse the node order in the incidence list
drncoors              *read the node coordinates
open                  *open the file with node elevations
 9 'elev.dat'
dfile
 9 '*' 5
drnelev               *read the interface 1 elevations
 1
drnelev               *read the interface 2 elevations
 2
drnelev               *read the interface 3 elevations
 3
```

```
drnelev             *read the interface 4 elevations
 4
*open               *open the file with node renumbering list
* 9 'roword.dat'
*rdroword           *read the node renumbering list
close               *close data file
 9
dfromc              *make the current command file also the data file
 '*' 1              *zfmt ngpl
```

In the above command file segment, two data files have been opened, using the same **open** command discussed before, and data is read from the file. Each time a data file is opened with a currently active unit number, the data file connected to the unit number is closed. So, although the unit numbers are reused, there is no conflict over where the data is coming from in this example. At the end of this command file segment, the currently open data file ("elev.dat") is explicitly closed using the **close** command.

The **dfile** command explicitly defines the unit number that data will be read from, and part of the format of the data. For the first **dfile** command, the data file unit is unit 9 (parameter *munit*), the data is in no particular format (a list-directed FORTRAN read will be used, for those familiar with FORTRAN), and there will be one group of data on each line of the data file (parameter *ngpl*). The second **dfile** command resets the expected number of groups of data expected for each line to 5. The **dfromc** command performs exactly the same function, except that the data file unit number is set to the current command file unit number, whatever it might be. These commands are companions to the **indexon** command, and work in exactly the same situations. Accordingly, these commands only have an effect for **drnxxxxx**, **denxxxxx**, **drexxxxx**, or **deexxxxx** commands encountered, where the **xxxxx** can refer to any of the distributed parameters.

The **dfile** command introduces a number of nuances best appreciated by those familiar with FORTRAN. In most cases, it will suffice to use the command as shown in the example above. As long as the data file is set up such that each line in the data file corresponds to data for one and only one node or element, there is little reason to use any other specification (except, of course, to change the unit number *munit*). The setup shown allows comments in the data, if desired, at the end of each line of the data file.

If the data file is set up so that there is more than one group of data on a line, for example if a table has been created, there are a couple of ways to handle the situation without modifying the data. If there are no comments in the data, the easiest thing to do is to change the *ngpl* parameter to -1. PTC then expects a stream of numbers, and there is no need for data for a particular node to be confined to one line of the input file. Data for more than one node can be put on a single line. Although the data can be spread over lines in any funky way, the numbers must be in order so they can be interpreted.

If one has comments in the data file, the previous strategy won't work. However, if the data is in tabular form, such that a set amount of groups of data exist on each line, by giving the *ngpl* parameter this number comments can be placed after the last group.

For those enamored with formatted FORTRAN input, your longings can be satisfied. Perhaps the best use that formatted input can be put to is to pick out a particular column of a large table. For example, if the nodal coordinates are in a file with lots of other extraneous information corresponding to the node, by specifying the appropriate format one can get just the coordinates. Say the coordinates are in the fourth and fifth data column, corresponding to the 31-40 fields and 41-50 fields in the data file. In this case, one could specify **dfile** as follows:

```
dfile
 9 '(30x,f10.0,f10.0)' 1
```

This skips the first 30 fields, then reads in two floating point numbers from 10 fields apiece. Note that this will only work if the data file is set up so that the nodal coordinates are in *exactly* the same place on each line. FORTRAN formats are discussed in more detail in the PTC Documentation and in the section on Graphical Output Files.

The **rdincid** command reads in the incidence list from the current data file ("mesh.dat"). In the simple example, the incidence list was generated using the **gnincid** command, where the incidence list is the node numbers comprising an element, specified in counter-clockwise order. In this case, a computer program generated the incidence list. The top portion of the "mesh.dat" file is as follows:

```
1  1 2 9 8
2  2 3 10 9
3  3 4 11 10
4  4 5 12 11
```

The file continues until all elements are specified.

The incidence list for a node requires four node numbers. PTC will handle either triangular or quadrilateral elements, although quadrilateral elements are considered more accurate. Triangles and quadrilaterals can both be specified in the same data file. A triangular element is signalled by setting the fourth node number to zero.

If the computer program uses the opposite convention for the incidence list than does PTC, and the nodes are specified in clockwise order, the **rvincid** command can be used to reverse the order of the nodes in incidence list. When the incidence list is opposite to the PTC convention, lots of error messages complaining about non-positive element areas will scream out when trying to run a **sim** command. In this example, the ordering is incorrect, so the command is used.

The **drncoors** command reads in the nodal coordinates for all of the nodes. In the simple example, a regular grid was generated using the **gnreccor** command. Both the node number and the coordinates are expected in the "mesh.dat" file, as signalled by the **indexon** command. The top portion of the "coors.dat" file is:

```
1 0. 0.
2 0. 57.142857
3 0. 114.28571
4 0. 171.42857
```

This continues until all nodes are specified.

The series of **drnelev** commands read in the elevation of the interfaces. All of the elevation data has been concatenated into one file; of course, there could be separate files for each interface. The “elev.dat” file consists of 76 lines, each with five elevations, and there are 4 groups of 91 nodes.

The **rdroword** command is used to externally specify the order the equations will occur in the horizontal coefficient matrices. This allows a bandwidth minimizer to calculate an efficient node ordering, without changing the node numbers used in specifying the rest of the data set. The default ordering is by node number. In this example, the input mesh is already optimal, so external reordering is not necessary.

6.3 Material Properties

For most of the material properties, the same procedure is followed for the complex problem as for the simple problem. Global properties are used for most of the properties. However, the hydraulic conductivity field for the clay layer is specified in more detail, due to improved field information on the position of the window. In this case, conductivity values are specified at nodal locations and corresponding elemental properties are calculated internally, using **drnxxxxx** commands. This procedure is often followed when using interpolation packages to specify a parameter field, as nodal coordinates are readily available. Were the elemental properties specified directly, **drexxxxx** commands would be used.

The portion of the command file which specifies the input of the material properties follows.

```
open
 9 'claycond.dat'
dfile
 9 '*' 1
drncondx            *read x conductivities
 2
rewind              *reset to the top of the file
 9
drncondy            *read y conductivities
 2
rewind
```

```
 9
scale
 0.1
drncondz            *read z conductivities
 2
scaleoff
```

In this example, layer 2 is the clay layer. They have the same x and y conductivity field (no anisotropy), and it is assumed that the z conductivity is a uniform 10% of the horizontal conductivity. Accordingly, the same data file is used to specify all of these values. The **rewind** command sets the data file to the beginning of the file, so that the same set of values can be input several times. Since the z conductivity is only a scaled version of the horizontal conductivity, the **scale** command comes in handy here.

6.4 Initial Conditions

The same initial conditions are used for the complex problem as are used for the simple problem.

6.5 Boundary Conditions

One last complexity of data input is introduced in this section. The boundary conditions corresponding to the river extend over several nodes. These boundary conditions are specified using a data generation facility. This data generation facility is only applicable when the the **indexon** command is active. The part of the command file specifying these boundary conditions is as follows:

```
*
* boundary conditions and forcing functions
*
indexon
dgenon
dfromc
 '*' 1
drnbcflo            *layer 3 Dirichlet bcs (river)
 3
  1  1    1 88.     *node, ng, flag, value
  5  1    1 91.
  7  0    1 92.
 0/
dgenoff
```

```
scale               *convert gpm to ft**3/day
192.5134
denbcflo            *layer 1 flux (pumping well) for flow
 1
 1   2 -10.
0/
scale               *convert in/year to ft/day
 2.283105e-4
drgrain             *10 in/year infiltration over top layer
 10.
scaleoff
denbccon            *layer 1 flux (pumping well) for flow
 3
 3   1 100.
 4   1 100.
0/
indexoff
```

The **dgenon** and **dgenoff** commands turn on and turn off the data generation facilities. The specification is exactly the same as occurs without data generation, except that a new column of generation codes appears after the node indices. In this example, each of the nodes between 85 and 89 get a value for Dirichlet head that is linearly interpolated between the values of 88 and 91, node 90 gets a value of 91.5, and node 91 gets a value of 92. Interpolation is signaled by a non-zero *ng* flag. In this case, as *ng* is 1, each node between the extremes is assigned an interpolated value. If *ng* were 2, every other node would get a value, and so on. The 0 value for *ng* signals that no interpolation is to be done; in order to list a set of node values, simply set all of the *ng* flags to 0.

Data generation is of most use when relatively regular features are present on regular parts of a mesh.

The remaining boundary conditions reprise the simple example commands.

6.6 Splitting Layers

The total number of layers that PTC expects are specified in the **rddims** command. There are 6 layers allocated, yet only the bottom three have had values assigned to them thus far. At this point, each of the three field layers are converted into a pair of computational layers.

When a splitting action is performed, PTC assumes that the top layers are redundant and can be discarded. For each new layer that is created, each overlying layer is pushed up one layer number, and the top layer information disappears. Since no information has been assigned to the top three layers, there is no loss in this splitting operation.

The following piece of the command file splits each field layer into two computational layers.

```
*
* convert field layers to computational layers
*
splayer              *split layer 3 into 2 layers
 3  2
splayer              *split layer 2 into 2 layers
 2  2
splayer              *split layer 1 into 2 layers
 1  2
dealeak              *aquitard below layer 1
 1  90. 1.e-5 -1.    *ref head, conductance, ref conc
desleak              *holding pond in layer 6
 6  96. .0001 20.    *ref head, conductance, ref conc
 72 79/
```

The splitting of field layers into computational layers takes place in three steps. The first **splayer** command makes layer 3 into layers 3 and 4, moving layer 4 to layer 5, layer 5 to layer 6, and discarding layer 6. Similarly, interface 4 is moved to interface 5, interface 5 is moved to interface 6, interface 6 is moved to interface 7, interface 7 is discarded, and interface 4 is assigned elevations midway between the interface 3 elevations and the new interface 5 elevations. This procedure is shown in Figure 2. The second and third **splayer** commands perform the same function for the clay layer and the lower sand layer. The final result is two sand layers, two clay layers, and two more sand layers, each with half the thickness of the parent layer.

Any Dirichlet or leakage conditions are copied to the new layers unchanged. Flux conditions are also copied to the new layers. Flux conditions, however, are divided by the number of new layers, and the mass entering the system at each node is distributed evenly over the number of new layers, so that the total mass entering the system remains unchanged. In this case, the previously specified pumping well remains fully penetrating, with half of the flux leaving from layer 1 and half from layer 2.

The river is assumed to fully penetrate the top sand layer, so the same boundary conditions are used in the top two layers, and the river conditions are applied *before* the splitting. On the other hand, the leakage from the bottom and the leakage from the holding pond are only applied to the lowermost and the uppermost layers; these conditions are applied *after* the splitting.

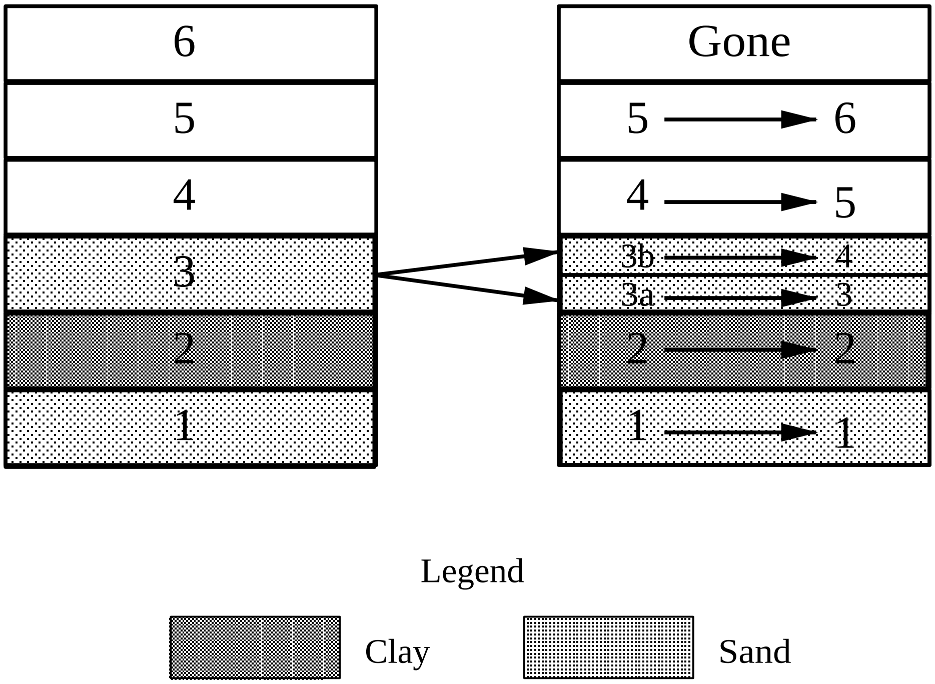

Figure 2: Splitting layer 2 into layers 2 and 3.

6.7 Simulation

The simulation of interest incorporates variable recharge rates, specified on a monthly basis, and is to occur over ten years. This could become tedious to specify in a data set using just the commands presented thus far. Two more of the PTC tools are used to ease this burden.

PTC allows control of the command input to be interactive. Similarly, separate files can contain pieces of a command session. By defining a separate file containing the commands for a full year's simulation, and reusing this file for each year, the input can be greatly reduced. By using the available looping facilities, the input is reduced further. The following part of the main command file performs the looping:

```
*
* loop over number of years
*
open
 8 ’yearsim.cmd’
do year             *begin loop ‘year’
 10
cfile               *change command input to unit 8
 8
rewind              *rewind unit 8
```

```
 8
od year             *end loop ‘year’
```

This simple construction opens the command file segment “yearsim.cmd”, and changes command input to the file segment using the **cfile** command. Upon return from the file segment, the file segment is rewound so that the file can be reused. This process is repeated 10 times, once for each of the ten years of the simulation, using the **doxxxxxx** command. The **odxxxxxx** command marks the end of the file segment that is looped over for label **xxxxxx**. In this case, label **xxxxxx** is “ year ”. Both the **cfile** and the **doxxxxxx** commands can be nested.

The following is the complete “yearsim.cmd” file.

```
open
 9 'yearrain.dat'
dfile
 9 '*' 1
domonth
 12
indexoff
scale               *convert in/year to ft/day
 2.283105e-4
drnrain
scaleoff
rdcontrl
 1 1 1 1 1          *yflow yvel yconc yincor ymbal
gntime
 10 5 50 1 1.5 30.42 *itmax itchng itchmx nmspf chng tleng
sim
odmonth
rewind
 9
close
 9
dreturn
creturn
```

This short file runs twelve combined flow and transport simulations. Each simulation has a different infiltration rate: the input for each of the twelve months have been concatenated into a single file “yearrain.dat”. Since the system doesn’t reach steady state quickly enough compared to the length of a simulation, the flow and transport simulators are run concurrently.

Concurrent flow and transport simulations are computationally demanding. Coefficient matrices must be factored every time step, which is an expensive operation. One should expect that such simulations will take several times longer than comparable simulations where the head field is allowed to reach steady state before transport begins, even for the same number of flow steps and transport steps.

The **rewind** and **close** commands reset the infiltration data file for the next invocation of the "yearsim.cmd" file. If the infiltration is different each month and each year, by skipping the **rewind** and **close** commands, and catenating all of the monthly data into "yearrain.dat", this same looping procedure can be used.

The **dreturn** and **creturn** commands are used to restore the command input state. The **dreturn** command restores the data file prior to issuing the previous **dfile** command. Similarly, the **creturn** command restores command file interpretation to the command file that was active immediately prior to issuing the previous **cfile** command, effectively returning to the "calling" command file. These commands hide actions that took place within the "subroutine" command file ("yearsim.cmd").

Admittedly, using the separate command file segment is somewhat contrived in this example. The idea is most useful when a particular command sequence is to be modified for different simulations, while the remainder of the command file is unchanged. In this case, saving only the modified command sequence, in a set of files, could be simpler than saving large input files with minor changes. These command sequences can act as "libraries", as well. One place where these libraries are handy are when specifying output formats for different graphics programs. This is discussed in detail in the section on Graphical Output Files.

7 Graphical Output Files

All of the output commands discussed thus far echo the input in a form for the human eye to read (tabular output, etc.). With the increasingly important use of post-processing, it is also imperative to create output in a form that other programs can use. A mechanism is provided in PTC so that any of the distributed variables can be output in a form selected by the user. An independent output stream for "graphical" output is provided. Detailed discussion of graphical output is presented in the documentation. The commands in this section are sufficient for most purposes; however, it is highly recommended that the appropriate section in the documentation is studied thoroughly, particularly if not familiar with FORTRAN formatting.

A sample of a data file which might be used with FEPER, a graphical package provided by Environ Corporation, is provided below.

```
*
* set up information previous to start of simulation.
```

```
*
gwrit
9
open
9 'cmdfil.fep'
*
grechon
newdim
200 0 0 0 4 40/
open
11 'feper.mac'/
lumacr
11 -1/
grechoff
*
* set up for feper output
*
grcoors
'(6hrdnod1/i5/6h-1 ''*''/(i4,t1,1pe16.8,1pe16.8))'
grincid
'(6hrdel1 /i5,2h 4/6h-1 ''*''/(i4,t1,4i5))'
grsolfmt
'(6hrdfun1/i5/6h-1 ''*''/(i4,t1,1pe12.4,t1,1pe12.4,t1,1pe12.4))'
rdgrfkey            *specify graphics output intervals
 50 50 50 50        *kfgbeg kfginc kmgbeg kmginc
```

In this section of the data file, the graphics output file is set to unit 9 (the **gwrit** command), with a name of "cmdfil.fep". The next command, **grechon**, signals that the next set of lines will be copied to the graphics output file uninterpreted, until a **grechoff** command is found. There is no limit to how many lines are in between the two commands. In this case, several FEPER commands are copied to the output file.

The **grcoors** command outputs the current nodal coordinates to the graphics file. The complex character string following this command is the format for the output. The format is used in a standard FORTRAN write statement, which outputs the number of nodes, and a string of node numbers with associated node coordinates. The format string above creates a valid FEPER input with the current mesh coordinates. This command has the effect of overwriting the node number, using the `i4,t1` construction. Note that a pair of quotes are needed for each quote desired in the format string. For more information on FORTRAN formats, consult the PTC documentation and a FORTRAN reference.

The **grincid** command outputs the current incidence list to the graphics output file. Again, the character string following the command is a format for the output. In this case, the write statement will output the number of elements, followed by a string of element number and associated incidences. Again, this structure creates a valid FEPER command.

The **grsolfmt** command defines the output format for solutions, including head, concentration, or other distributed values. Any of the distributed values can be output to the graphics file, by issuing a **grxxxxxx** command instead of a **ecxxxxxx** command for distributed parameter **xxxxxx**. This provides a convenient way to check that fields are defined appropriately. The write statement for one of these solutions includes a function number, which is incremented for each layer output, and a series of node number, x coordinate, y coordinate, and function value. In this example, the x coordinate and y coordinate are overwritten by the solution.

In the same way that the **rdprtkey** controls the output of solutions to the results output file(s), the **rdgrfkey** command controls the output of solutions to the graphical output file.

Since the graphics commands are usually the same from data set to data set, and are somewhat complicated, it is a good strategy to create a file containing only graphics commands. Using the **cfile** command, as when performing the yearly simulations in the complex example, these commands can be used without including them in the command file. As a result, these commands are "hidden" from the casual user.

8 Debugging a Data Set

Sooner or later, a simulation will act badly. The head field will go haywire, contaminants will pop up in weird places, or PTC will just stop dead. These types of problems are inevitable, particularly in early stages of creating a data set.

In order to help diagnose the cause of such problems, there are a number of internal checks that are performed immediately prior to a simulation. The input mesh is checked for plausibility, and all material properties are checked to see if they have positive values. If there is a problem, the error is reported to the output file (**owrit** file) and PTC stops immediately. When PTC stops much sooner than expected, the first thing to look for is one of these error messages in the output file. Such problems are usually easy to remedy.

If PTC doesn't catch these errors, however, problems are more difficult to detect. In some cases, inspecting echoed data will reveal that some variable has values that are just not the values that were entered. A common mistake is to clear an array, when it should be editted. In this case, there should be a large number of zero values.

In other cases, values that were supposed to go into one array end up in another array. In this case, the indexing of the first array is a prime suspect. For example, if there are 20 nodes dimensioned for a problem, and the indexing has a node 40, the offending value is put somewhere other than within the array of interest.

If a command is not recognized, through misspelling perhaps, an error message is output to that effect and the command is ignored. All of the data associated with the command is presumed to be a command, and a list of these error messages can result.

If not enough data is provided for a command, subsequent commands can get "swallowed" as data, with bizarre effects. If too much data is provided for a command, a series of "Command not found" messages can result.

Sometimes it is apparent that there is some sort of problem, but it is not clear where the problem is created. In such cases, the interactive facilities that PTC provides become extremely useful. The **term** command is used to enter interactive mode, and the **go** command is used to leave interactive mode. Any of the PTC commands can be issued from the interactive mode; the echo commands are especially useful. By littering **term** commands throughout the command file, and echoing the commands to the terminal (unit 6), the progress of the data set can be monitored easily. The value of any variable can be determined at each pause. Usually a bisection approach will rapidly locate the part of the command file that is causing the problem.

In larger data files, one of the best ways of verifying that data has been read in correctly is to actually plot the data values. Each of the distributed variables can be output for a graphics package to plot. This is discussed in greater detail in the section on Graphical Output Files.