NISTIR 5612

4SIGHT Manual: A Computer Program for Modelling Degradation of Underground Low Level Waste Concrete Vaults

Kenneth A. Snyder
James R. Clifton

June 1995
Building and Fire Research Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

U.S. Department of Commerce
Ronald H. Brown, *Secretary*
Technology Administration
Mary L. Good, *Under Secretary for Technology*
National Institute of Standards and Technology
Arati Prabhakar, *Director*

Abstract

A computer program has been written to facilitate performance assessment of concrete vaults used in Low Level Waste (LLW) disposal facilities. The computer program is a numerical computer model of degradation in concrete. A one-dimensional finite difference equation is used to propagate ions by precipitation/dissolution of available salts. The precipitation/dissolution of salts, in turn, changes the transport properties, which changes the rate of ion transport. The result is a model which incorporates the synergism of multiple degradation mechanisms.

This Report is self-contained. It includes the installation instructions, user manual, technical details, and source code. The program was written using a literate programming tool and the "pretty-printing" output of the source code is attached at the end of this report.

Keywords: building technology; computer modelling; concrete; corrosion of reinforcement; degradation; leaching; radioactive waste; service life; sulfate attack

Contents

List of Figures

1 Introduction

The computer program 4SIGHT has been written to facilitate Low Level Waste (LLW) disposal facility performance assessment. 4SIGHT can be used to predict the service life and hydraulic conductivity of a buried concrete vault. This estimate is based upon an analysis of the concrete vault roof given the material properties of the concrete and the external ion concentrations.

4SIGHT incorporates multiple degradation mechanisms by using a single transport equation for ions. Individual degradation mechanisms are typically controlled by the concentration of a single ion species. However, as various ion species diffuse into the concrete, the transport properties change, which changes the rate of ion diffusion. As the pore solution pH changes, available salts either precipitate or go into solution. The precipitation and dissolution processes changes the porosity, which in turn changes the transport properties. It is the act of maintaining chemical equilibrium that affects the synergism of the different degradation mechanisms.

Previous efforts [1][2] have yielded mathematical models for various degradation mechanisms. From these equations, the service life could be calculated for any one degradation mechanism. However, combining the effects of multiple mechanisms was not feasible analytically. The advantage of the method used by 4SIGHT is the ability to incorporate the synergism of multiple degradation processes. Rather than calculate rates of attack, 4SIGHT uses a single transport equation to propagate ions through the concrete. Keeping the system in chemical equilibrium allows interaction of degradation mechanisms.

4SIGHT is a command-driven program that awaits input parameters from the user, using default values when necessary. Once all known parameters are entered, 4SIGHT reiterates the values of all the input parameters used in the calculation with a note as to whether the value was specified by the user or whether a default value was used. The computations begin by 4SIGHT propagating ions in discrete time increments. At regular intervals 4SIGHT prints to the screen the bulk hydraulic conductivity and diffusivity, and the depth of sulfate and chloride attack. The calculation continues until one of the following occurs: corrosion of the steel reinforcement, sufficient sulfate attack that the roof fails structurally, failure of a joint, or a user specified time limit has been exceeded. Upon termination, 4SIGHT prints the state of the system, composed of the concentration of user selected ions and moles of user selected salts, as a function of depth into the roof.

What follows in this Report is a complete user's guide to 4SIGHT, including installation information and source code. The goal is an all-inclusive document, acting as a single body of work, that facilitates review of any aspect of the program. To facilitate source code review, 4SIGHT was written using a literate programming tool which incorporates both the programming code and typeset comments. The typeset output does not conform to the structure of a NISTIR. Therefore, the typeset output, as it normally appears, is attached to the end of this report.

2 Installation

2.1 Distributed Files

4SIGHT is distributed with the following files:

README	Basic installation instructions
4SIGHT.EXE	Executable program
ION.DB	Ion database
MANUAL.PS	4SIGHT manual in PostScript format
MANUAL.HP	4SIGHT manual in HP-PCL format
MANUAL.TXT	4SIGHT manual in ASCII format
EXAMPLE.DAT	Example input file

To install 4SIGHT, simply copy these files to a directory on the hard drive. This manual has been included in three different useful formats for printing.

2.2 Hardware Requirements

To execute the program 4SIGHT, the user must be using a personal computer equipped with either an 80386 or 80486 microprocessor with a math coprocessor. Lacking either of these requirements, the program will fail to run.

3 Program Execution

4SIGHT was written to be executed in either interactive or batch mode. The interactive mode simply waits for the user to type in the various input parameters at run-time. When executed in batch mode, the user first edits an ASCII file containing the input parameters as if typed during interactive mode. This ASCII file is then included at the command line when running 4SIGHT.

3.1 Interactive Mode

When run interactively, 4SIGHT simply waits for the user to enter input parameters. The user specifies the end of the list by entering either `quit` or `exit`. For example, assuming the executable program is in the directory `C:\4SIGHT`, interactive mode is initiated by simply typing the program name:

```
C:\4SIGHT\4SIGHT
```

The program responds with:

```
Enter commands:
```

At this point 4SIGHT is simply waiting for the user to input program parameters. The commands are given one per line:

```
DIFF = 2.0E-12
WC = 0.45
TIME = 100000
quit
```

After which, 4SIGHT begins the calculation, printing intermediate results to the screen.

3.2 Batch Mode

When run in batch mode the user specifies an input file at the command line. This input file simply contains any number of input parameters, as in the interactive mode. For example, assume the following text is saved into a file called `input.dat`:

```
DIFF = 5.0E-12
THICKNESS = 1.0
WC = 0.50
EXTERNAL Cl = 0.150
EXTERNAL Na = 0.150
DEPTH = 0.25
```

This file can be created by any ASCII editor such as the DOS `edit` program or the UNIX `vi` program. When using a commercial word processing program to create input files make sure that the output is in plain ASCII. As a check, give the `C:\type input.dat` command at the DOS prompt. The text should appear as typed. Note that neither `quit` nor `exit` are required in the input file. To use input files with 4SIGHT, simply include that file name on the DOS command line:

```
C:\4SIGHT\4SIGHT input.dat
```

The output from 4SIGHT goes to the screen. This is useful for initial calculations and trial runs. However, if the user desires to save the output to an ASCII file, simply use redirection of the output:

```
C:\4SIGHT\4SIGHT input.dat >output.dat
```

This example uses the commands in the file `input.dat` as input parameters and saves the output in the file `output.dat`.

3.3 Plotting Results

The output from 4SIGHT has been formatted with quotes (") and tabs to facilitate incorporating them into commercial spreadsheet programs. Once incorporated into the spreadsheet program, the user can plot either the sulfate and chloride depth versus time from the intermediate results, or the ion concentration as a function of depth from the final state of the system.

4 Input Specifications

The complete list of input parameters is given here to introduce the parameters used in the example to follow. Although many of the parameters are self-evident, the complete meaning of all the parameters will not be understood until the user reads the explanatory sections that follow the example.

The input commands to 4SIGHT are straightforward. With very few exceptions, the format of the commands is

parameter = *value*

Even though every possible input parameter is listed below, there are sufficient default parameters for the user to perform a calculation without entering any parameters.

The syntax of the list of parameters below is as follows: Text to be typed verbatim is shown in typewriter font (*e.g.*, `DIFF =`). These words can be typed in any mixture of upper and lower case characters with any number of spaces or tabs between the parameters and the '='. Words in italics represent user selected input values (*e.g.*, *expr*). There are two data types: numbers and ions, which are represented by *expr* and *ion*, respectively. Numbers can be given in integer, fixed point, or scientific notation. Examples of valid values for *expr* include:

```
120      120.0      1.2E+02
```

The units for each input value appear under *expr* in the definition. 4SIGHT also recognizes the following ions (the valence has been omitted):

```
H      Ca      Na      K      OH      Cl      SO4      CO3
```

The ion values MUST appear as shown since ion variables are case sensitive. 4SIGHT can also recognize salts. A salt is specified by concatenating a cation and an anion, separated by a space. Stoichiometric ratios are not used. For example, the following specifications

```
Na Cl          Ca OH          Na SO4
```

are for sodium chloride, $NaCl$, calcium hydroxide, $Ca(OH)_2$, and sodium sulfate, Na_2SO_4, respectively.

Below is a list of all possible input parameters. At the end of each description is an example of usage. In cases where a default value exists, the default value is used as the example value. Whether or not the value is a default value is noted next to the example.

4.1 Material Properties

`DIFF =` *expr*
m^2/sec

expr: The ultimate Cl^- diffusivity of the concrete. This value represents the diffusivity of the undamaged concrete. The default value given below is suitable for a w/c=0.45 ordinary portland concrete.
default: `DIFF = 5.7E-12`

`PERM =` *expr*
m^2

expr: The ultimate permeability of the concrete. This value represents the permeability of the undamaged concrete.
default: `PERM = 2.5E-18`

`WC =` *expr*

expr: Concrete water:cement weight ratio during mixing.
default: `WC = 0.45`

4.2 Ion/Fluid Initial Conditions

`EXTERNAL` *ion* `=` *expr*
mol/L

ion: The ion of interest.
expr: The external concentration of *ion* above the roof. This command can be repeated for each ion external to the roof slab.
example: `EXTERNAL Cl = 0.15`

`INTERNAL` *ion* `=` *expr*
mol/L

ion: Ion of interest.

expr: The concentration of *ion* in the pore solution of the concrete. This command was designed to let the user specify the internal Na^+ and K^+ concentration to establish the correct pH. This command can be repeated for each relevant ion.

example: `INTERNAL Na = 0.10`

`HEAD =` *expr*
m

expr: The equivalent height of ground water (hydraulic head) above the upper surface of the roof.

default: `HEAD = 5.00`

4.3 Geometry.

`THICKNESS =` *expr*
m

expr: The vertical thickness of the roof.

default: `THICKNESS = 1.00`

`CRACK =` *expr1* `AT` *expr2* `DEPTH` *expr3*
m *m* *m*

expr1: The width of the crack.

expr2: The spacing between cracks.

expr3: The penetration of the crack, relative to the bottom of the slab.

default: `CRACK = 0.000100 AT 1.00 DEPTH 0.30`

`JOINT PERM =` *expr*
m^2

expr: The permeability of the joint filling compound between roof slabs. This value should not reflect the geometry of joint. Rather, it is a material property of the joint compound.

example: `JOINT PERM = 3.2E-15`

`JOINT =` *expr1* `AT` *expr2* `UNTIL` *expr3*
m *m* *years*

expr1: The width of the joint.
expr2: The spacing between joints.
expr3: The service life of the joint compound.
example: `JOINT = 0.020 AT 1.00 UNTIL 100`

4.4 Failure/Termination Limits

`REBAR =` *expr*
m

expr: The depth of the reinforcement bars (rebars) from the top of the slab. When a critical concentration of Cl^- ions has penetrated to a depth *expr* the structure fails.
example: `REBAR = 0.90`

`DEPTH =` *expr*
m

expr: The critical penetration depth for sulfate attack. Once the sulfate front has penetrated down to a depth *expr* from the top, the roof is unable to sustain its load and it fails.
default: `DEPTH = 0.20`

`TIME =` *expr*
day

expr: The maximum number of days to continue the calculation. Note that no comma should be used.
default: `TIME = 100000`

4.5 Sulfate Attack Parameters

`YOUNGS =` *expr*
GPa

expr: The Youngs modulus of the concrete.
default: `YOUNGS = 20.0`

`BETA =` *expr*

expr: The linear strain due to one mole of Na^+ per m^3 of concrete.
default: `BETA = 1.8E-06`

`CE =` *expr*
mol/m^3

expr: The concentration of reacted sulfate as ettringite.
default: `CE = 350.0`

`ROUGHNESS =` *expr*

expr: The fracture surface roughness factor.
default: `ROUGHNESS = 1.0`

`GAMMA =` *expr*
J/m^2

expr: The fracture surface energy of the concrete.
default: `GAMMA = 10.0`

`POISSON =` *expr*

expr: The Poisson ratio of the concrete.
default: `POISSON = 0.2`

4.6 Output Parameters

`OUTPUT =` *ion*

ion: Output the pore solution concentration (mol/L) of *ion* in the output describing the final state of the system. This command lets the user examine the concentration of specified ions as a function of depth. This command can be repeated for different ions.
example: `OUTPUT = CO3`

`OUTPUT =` *ion1 ion2*

Output the quantity (moles) of salt in the volume `Vsample`. The salt is comprised of the cation *ion1* and the anion *ion2*.

NOTE: The two ion specifications are separated by a space and stoichiometric ratios are omitted.
example: `OUTPUT = Ca OH`

5 Example

To illustrate the use of 4SIGHT, consider the following fictitious example. The concrete was designed with a 0.40 water:cement ratio (WC=0.40). Experimental diffusivity measurements using Cl^- ions gave $5.0 \times 10^{-12} m^2/sec$ (DIFF=5.0E-12). No permeability measurements are available. The roof slab is 1.0 meter thick (THICKNESS=1.000). Regularly spaced cracks have been observed on the bottom of the slab. The cracks are approximately 100 μm wide, spaced 2 meters apart, and are assumed to extend upwards to the neutral axis which is 25 centimeters from the bottom of the roof slab (CRACK = 0.000100 AT 2.0 DEPTH 0.25). Also, the roof is buried, giving an effective pressure head of 2 meters (HEAD = 2.0).

Soil analysis indicates the presence of SO_4^{2-} at a concentration of 1.0 moles per liter (EXTERNAL SO4 = 1.0). Engineering analysis indicates that if the sulfate degradation penetrates down 20 centimeters from the top surface of the roof then the vault will collapse (DEPTH = 0.20). Additionally, chloride ions are present in the soil at a concentration of 0.40 moles per liter (EXTERNAL Cl = 0.40) and engineering drawings indicate that the reinforcement bars are located 60 centimeters from the top of the roof (REBAR = 0.60). Thorough soil analysis indicates that sodium ions are also present at a concentration of 2.40 moles per liter (EXTERNAL Na = 2.40), giving a nearly neutral soil pH.

Internal to the concrete the pH is approximately 13. Therefore, internal potassium and sodium concentrations are approximately 0.1 (INTERNAL K = 0.1) and 0.05 (INTERNAL Na = 0.05) moles per liter, respectively.

To monitor the ingress of the external ions, calcium (OUTPUT Ca) and chloride (OUTPUT Cl) ions will be included in the output of the final state of the system. Also, to monitor leaching, the solid calcium hydroxide content will also be included in the output (OUTPUT Ca OH).

The input file for this example, `example.dat` is included in the distribution diskette and reiterated here:

```
DIFF = 5.0E-12
WC = .40
THICKNESS = 1.000
EXTERNAL Na = 2.40
EXTERNAL Cl = 0.400
EXTERNAL SO4 = 1.00
INTERNAL K = 0.1000
INTERNAL Na= 0.0500
OUTPUT Ca
OUTPUT Cl
OUTPUT Ca OH
REBAR = .8000
```

```
HEAD = 2.0
CRACK = 0.000100 AT 2.0 DEPTH 0.25
DEPTH = .2000
TIME = 100000
```

To use this input file, simply include `example.dat` at the DOS command line. To save the results, redirect the output to a file:

```
C:\4SIGHT\4SIGHT EXAMPLE.DAT >EXAMPLE.OUT
```

In this example, the output is stored in `EXAMPLE.OUT`, an ASCII file which can be imported into any spreadsheet program.

NOTE: This calculation required 90 seconds to complete on a personal computer equipped with a 80486 microprocessor operating at 66MHz.

5.1 Internal Parameters.

4SIGHT outputs all of the parameters the user has specified ("USER"), along with default and calculated values ("DEFAULT").

```
"       This is 4SIGHT (Version 1.0)"

"THICKNESS "   1.00000 "(m)       " "USER"
"DIFF      "   5.0e-12 "(m^2/sec)" "USER"
"PERM      "   9.8e-13 "(m/sec)   " "DEFAULT"
"WC        "   0.40000 "          " "USER"
"HEAD      "   2.00000 "(m)       " "USER"
Sulfate Attack Parameters:
"YOUNGS    "   2.0e+10 "(N/m^2)   " "DEFAULT"
"BETA      "   1.8e-06 "          " "DEFAULT"
"CE        "  350.00000 "(Mol/m^3)" "DEFAULT"
"ROUGHNESS "   1.00000 "          " "DEFAULT"
"GAMMA     "  10.00000 "(J/m^2)   " "DEFAULT"
"POISSON   "   0.20000 "          " "DEFAULT"

"DEPTH     "   0.20000 "(m)       " "USER"
"REBAR     "   0.80000 "(m)       " "USER"
"TIME      "    100000 "(day)     " "USER"
"CRACK =  0.00010 AT  2.00000 DEPTH  0.25000"
"Chloride failure (yr)"      1743
"Sulfate failure (yr)"       447
```

5.2 Initial State

The report of the initial state lets the user verify the EXTERNAL and INTERNAL conditions of the system.

```
Initial state of system:

" ION " "EXTERNAL" "INTERNAL"
"   H:"  0.00000  0.00000
"  Ca:"  0.00000  0.00035
"  Na:"  2.40000  0.05000
"   K:"  0.00000  0.10000
"  OH:"  0.00000  0.15070
"  Cl:"  0.40000  0.00000
" SO4:"  1.00000  0.00000
"  pH:"  7.00000 13.17810
```

These results indicate that the pH of the environment is 7, while the pH of the pore solution is initially 13. To change the pH of the environment the user can simply change the concentration of EXTERNAL anions or cations.

5.3 Depth vs. Time

As 4SIGHT is calculating ion transport it regularly prints the current status of the degradation. The sulfate and chloride penetration depths as a function of time are given in the columns labelled SO4 and Cl, respectively. L is the remaining thickness of the slab, K is the hydraulic conductivity, D is the diffusivity, Flux is the flux of pore solution out the bottom of the slab, and pH is the pH of the flux.

```
"Day"   "L"      "K  "   "D     " "SO4"   "Cl"    "Flux"   "pH"
"   "   "m"      "m/s"   "m^2/s" "m  "   "m "    "ml/dy/m2"

0         1.000 1.3e-12 5.0e-12  0.000  0.000  0.027   13.2
10036    0.988 1.3e-12 5.0e-12  0.012  0.101  0.027   13.2
15055    0.982 1.3e-12 5.0e-12  0.018  0.129  0.027   13.2
20023    0.976 1.3e-12 5.0e-12  0.025  0.150  0.028   13.2
25031    0.969 1.3e-12 5.0e-12  0.031  0.173  0.028   13.2
30018    0.963 1.3e-12 5.0e-12  0.037  0.192  0.028   13.2
35011    0.957 1.3e-12 5.0e-12  0.043  0.211  0.028   13.2
40048    0.951 1.3e-12 5.0e-12  0.049  0.229  0.029   13.2
45104    0.945 1.3e-12 5.0e-12  0.055  0.246  0.029   13.2
50017    0.939 1.3e-12 5.0e-12  0.061  0.263  0.029   13.2
55045    0.933 1.3e-12 5.0e-12  0.067  0.280  0.029   13.2
60015    0.927 1.3e-12 5.0e-12  0.074  0.295  0.030   13.2
65003    0.920 1.3e-12 5.0e-12  0.080  0.311  0.030   13.2
70002    0.914 1.3e-12 5.0e-12  0.086  0.327  0.030   13.2
75013    0.908 1.3e-12 5.0e-12  0.092  0.342  0.031   13.2
80069    0.902 1.4e-12 5.0e-12  0.098  0.357  0.031   13.2
85037    0.896 1.4e-12 5.0e-12  0.104  0.372  0.031   13.2
90050    0.890 1.4e-12 5.0e-12  0.110  0.387  0.031   13.2
95069    0.884 1.4e-12 5.0e-12  0.116  0.401  0.031   13.2
100017    0.878 1.4e-12 5.0e-12  0.122  0.416  0.032   13.2
```

As per the linear model for sulfate attack, the sulfate front increases linearly with time. The chloride depth has a $t^{1/2}$ because of the low Peclet number (the ratio of Darcy to 'diffusive' flow [2]). At sufficiently high Peclet numbers (greater permeability or hydraulic head) the chloride depth approaches a linear relationship to time.

5.4 Failure Data

After reporting the time dependent behavior, 4SIGHT reports the reason for termination and the status of the sulfate and chloride penetration:

```
Exceeded TIME limit.

"T"    17.283
"Day"    100017
"SO4 (m)"     0.122
"Cl (m)"     0.416
```

These failure data indicate that the calculation terminated because it exceeded the time limit. The termination occurred after 100044 days (586 years), at which point the sulfate and chloride penetration were 0.13 and 0.60 meters, respectively.

5.5 Final System State

Upon termination of the calculation, the final state of the system for the slab is printed. The `L(m)` is measured from the top of the slab. `Psi` is the hydraulic pressure, `vD` is the Darcy velocity, `xi` is the inverse of the formation factor, `phi` is the porosity, `fc` is the estimated compressive strength using ACI 211.

```
Final System state:

"L(m)" "Psi" "vD"  "xi"   "phi"  "pH"  "fc" "Ca"   "Cl"   "CaOH"
0.0000 0.392 0.000 0.0024 0.1634  7.000 5052 0.0000 0.4000 0.000
0.0500 0.392 0.000 0.0024 0.1631  7.000 5057 0.0000 0.4000 0.000
0.1000 0.392 0.031 0.0024 0.1658  7.000 5011 0.0000 0.4000 0.000
0.1500 0.374 0.031 0.0024 0.1663 13.277 5003 0.0001 0.3740 31.599
0.2000 0.342 0.031 0.0025 0.1671 13.351 4989 0.0001 0.3253 31.860
0.2500 0.311 0.031 0.0025 0.1672 13.392 4988 0.0001 0.2773 31.886
0.3000 0.279 0.031 0.0025 0.1672 13.405 4987 0.0001 0.2314 31.899
0.3500 0.248 0.031 0.0025 0.1673 13.394 4986 0.0001 0.1889 31.912
0.4000 0.217 0.031 0.0025 0.1673 13.365 4986 0.0001 0.1509 31.918
0.4500 0.186 0.031 0.0025 0.1673 13.326 4986 0.0002 0.1178 31.919
0.5000 0.155 0.031 0.0025 0.1673 13.286 4986 0.0002 0.0899 31.919
0.5500 0.124 0.031 0.0025 0.1673 13.250 4986 0.0002 0.0671 31.919
0.6000 0.093 0.031 0.0025 0.1673 13.220 4986 0.0003 0.0488 31.919
0.6500 0.062 0.031 0.0025 0.1673 13.199 4986 0.0003 0.0348 31.919
0.7000 0.031 0.031 0.0025 0.1673 13.185 4986 0.0003 0.0242 31.919
0.7500 0.000 0.031 0.0025 0.1673 13.176 4986 0.0004 0.0164 31.919
0.8000 0.000 0.031 0.0025 0.1673 13.173 4986 0.0004 0.0110 31.919
0.8500 0.000 0.031 0.0025 0.1673 13.172 4986 0.0004 0.0073 31.919
0.9000 0.000 0.031 0.0025 0.1673 13.172 4986 0.0004 0.0050 31.919
0.9500 0.000 0.031 0.0025 0.1673 13.172 4986 0.0004 0.0039 31.919
1.0000 0.000 0.031 0.0025 0.1673 13.172 4986 0.0004 0.0039 31.919
```

Note that the external quantities are duplicated to a depth of 0.1 meters. As the sulfate penetrates the concrete, the concrete fails and the external boundary conditions move into the concrete. Also note that the pressure (`P`) is zero from 0.75 to 1.00 meters. Since the concrete was cracked, the permeability of the cracked portion of the concrete overwhelmed

the uncracked portion, resulting in virtually no pressure drop across the crack. This increases the pressure gradient across the remaining uncracked concrete.

6 System

Although each LLW facility may be unique, most underground facilities can be represented schematically as in Figure 1. Below the ground surface there will be an engineered barrier to deflect surface water away from the concrete vault below. The entire facility is sited on a geologically suitable location.

Figure 1: Schematic of underground LLW facility

From the schematic in Figure 1 it appears as though the roof is the most critical element because it is most likely to have a moist environment, especially upon the failure of the engineered barrier. Because of this, the analysis of the entire concrete vault can concentrate on the roof. If the roof is treated as a simple slab, the analysis simplifies further. Since the flow through the roof slab will be approximately uniform over the surface of the slab, a one-dimensional analysis of transport vertically through the slab should serve as a sufficiently accurate model for transport through the roof of the vault.

7 Ion Transport

A single transport equation is developed to propagate ions through the slab. This equation can be converted into a finite difference equation so that it can be implemented in a computer program. A result of this approach is that time will advance in discrete intervals. After every time interval each computational element is put in chemical equilibrium using solubility products and a charge balance. This step is achieved through dissolution/precipitation of available salts. Any change in the quantity of solid salts in the pore space will change the

porosity of the concrete, hence changing the transport coefficients. This is how the synergism of degradation mechanisms is achieved.

7.1 Advection-Diffusion

At the core of 4SIGHT is the advection-diffusion equation which establishes the transport of ions through the slab. The advection-diffusion equation is simply the diffusion equation with an extra term to account for Darcy flow. The development of the equation starts most simply from the relation between flux, $\mathbf{j}$, and concentration, c:

$$\mathbf{j} = -D\nabla c + c\mathbf{u} \tag{1}$$

The parameter D is the effective diffusion coefficient, and the quantity $\mathbf{u}$ is the average pore fluid velocity. The rate of change in concentration is the negative divergence of eqn. 1:

$$\frac{\partial c}{\partial t} = \nabla \cdot D\nabla c - \mathbf{u} \cdot \nabla c \tag{2}$$

after neglecting the divergence of the volume averaged velocity, since the fluid is virtually incompressible and the rate change in porosity is small.

7.2 Darcy Flow

The average pore fluid velocity can be calculated from the Darcy velocity, which is the volume-averaged pore fluid velocity. Given a porous media with permeability k and pore fluid viscosity μ, the Darcy velocity, $\mathbf{v}_D$, is proportional to the pressure gradient, ∇p and the density of the fluid [3]:

$$\mathbf{v}_D = -\frac{k}{\mu}(\nabla p - \rho \mathbf{g}) \tag{3}$$

The Darcy velocity $\mathbf{v}_D$ can be related to the average pore velocity, $\mathbf{u}$, from a geometrical argument. Let $\mathbf{v}(\mathbf{x})$ represent the velocity of the pore fluid at some point $\mathbf{x}$ in the slab. Assume that the pore fluid completely fills the available pore space. Representing the porosity by ϕ, $\mathbf{u}$ is defined as:

$$\mathbf{u} = \frac{1}{\phi V}\int_{\phi} \mathbf{v}(\mathbf{x})\, dV \tag{4}$$

The Darcy velocity is the average over the entire volume V:

$$\mathbf{v}_D = \frac{1}{V}\int_{V} \mathbf{v}(\mathbf{x})\, dV \tag{5}$$

Since $\mathbf{v}(\mathbf{x}) = 0$ outside of the porosity, eqn. 5 can be limited to the pore space:

$$\mathbf{v}_D = \frac{1}{V}\int_{\phi} \mathbf{v}(\mathbf{x})\, dV \tag{6}$$

From comparison to eqn. 4, the relation between $\mathbf{v}_D$ and $\mathbf{u}$ is:

$$\mathbf{v}_D = \phi \mathbf{u} \tag{7}$$

7.3 Continuity

Once the Darcy equation is incorporated into eqn. 2, the continuity equation is needed in order to update the pressures. Upon execution of the computer program, the following sequence is continuously reiterated: transport ions, attain chemical equilibrium, adjust transport properties, update boundary conditions. As the porosity of the concrete changes, the transport properties also change. The change in the transport properties will effect the hydraulic pressure distribution in the concrete since the transport properties will not by uniform throughout the concrete. The pressure distribution will be updated using the continuity equation for porous media.

The continuity equation for a fluid is

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \mathbf{v} = 0 \tag{8}$$

To develop a continuity equation for porous media, find the average integral value of eqn. 8 over a representative volume, V:

$$\frac{1}{V}\int_V \frac{\partial \rho}{\partial t}\, dV + \frac{1}{V}\int_V \nabla \cdot \rho \mathbf{v}\, dV = 0 \tag{9}$$

Rearranging the order of integration gives

$$\frac{\partial}{\partial t}\frac{1}{V}\int_V \rho\, dV + \nabla \cdot \frac{1}{V}\int_V \rho \mathbf{v}\, dV = 0 \tag{10}$$

Finally, assuming ρ is constant and using the definition of Darcy velocity gives

$$\frac{\partial \phi}{\partial t} + \nabla \cdot \mathbf{v}_D = 0 \tag{11}$$

This is the result obtained by Slattery [4] for porous media. Substituting for $\mathbf{v}_D$ from eqn. 3 gives

$$\frac{\partial \phi}{\partial t} = \nabla \cdot \frac{k}{\mu}(\nabla p - \rho \mathbf{g}) \tag{12}$$

7.4 Dimensionless Variables

Eqn. 2, 3, and 11 can be combined to form a system of equations to propagate ions through a porous media. However, the system of equations can be condensed by a transformation into dimensionless variables. Consider the following definitions:

$$X = \frac{x}{L} \qquad T = \frac{t}{k_o/D_o} \qquad P = \frac{p}{\mu D_o/k_o} \tag{13}$$

where the capital letters X, T, and P refer, respectively, to the dimensionless distance, time, and pressure. The quantity D_o is the chloride diffusivity of the concrete, and k_o is the permeability. L is simply any characteristic length. Using these definitions, the advection-diffusion equation (2) becomes

$$\frac{\partial c}{\partial T} = \nabla \cdot \frac{D_i}{D_o} \nabla P + \frac{k}{k_o} \mathbf{v}_D \cdot \nabla c \tag{14}$$

This equation will be simplified further later using material properties related to porosity.

7.5 Atkinson-Hearne Model

As yet, a dependable mechanistic model does not exist for sulfate attack. Therefore, to incorporate sulfate attack, the depth of sulfate degradation is calculated using the Atkinson and Hearne [5] model:

$$R = \frac{E\beta C_o C_E D}{\alpha\gamma(1-\nu)} \tag{15}$$

E	Youngs modulus
C_o	External sulfate concentration
C_E	Concentration of sulfate as ettringite
D	Sulfate diffusion coefficient
α	Roughness factor
γ	Fracture surface energy
ν	Poisson ratio

Since the Atkinson-Hearne model gives the location of the sulfate front, `4SIGHT` presumes that all of the concrete behind the sulfate front has been completely disintegrated, giving it the properties of the surrounding soil. Since the transport coefficients of soil are much larger than concrete, the external boundary conditions are advanced to the sulfate front, creating a moving boundary condition.

The Youngs modulus, roughness factor, fracture surface energy, and Poisson ratio must be determined from experimental measurements. The quantity C_E can be either calculated from the cement composition, or more accurately from experiment. To experimentally determine C_E, the sulfate reacted per unit mass hydrated cement is plotted against the logarithm of time. This data is fit to the equation [5]

$$m = m_o \log_{10}\left(\frac{tc}{t_r c_k}\right) \tag{16}$$

where m is the moles of sulfate reacted in the cement, m_o is the free parameter of the regression, t is time, c is the concentration of sulfate in liquid, t_r is the characteristic time for reaction, and c_k is the concentration in kinetic experiments. The maximum value of m

can be calculated from the initial quantity of C3A in the cement. If this maximum value is labelled m_c, the value of t for $m = m_c$ is the time at which all of the C3A is consumed, $t = t_\infty$.

The model proposed by Atkinson and Hearne assumes that reaction is the controlling rate process. Therefore, the time to spalling should be greater than the time required for complete consumption of the C3A. The time to spalling is [5]

$$t_{spall} = \frac{X_{spall}^2 C_E}{2Dc_o} \tag{17}$$

where

$$X_{spall}^2 = \frac{2\alpha\gamma(1-\nu)}{E(\beta C_E)^2} \tag{18}$$

In the event that $t_{spall} < t_\infty$, C_E must be calculated in a self consistent manner using eqns. 16,17, and 18. More complete details can be found in [5]. For ordinary portland cements, it is likely that $t_{spall} > t_\infty$. Therefore, since it is assumed that the quantity of external sulfate is sufficiently great to act as an infinite reservoir (external concentration is constant), C_E can be calculated from the C3A content of the cement. Since each mole of ettringite requires one mole of Al_2O_3[6], the molar concentration of Al_2O_3 will be the molar concentration of ettringite. Given a concrete mix design having x_{cem} kilograms of cement per cubic meter of concrete, and the cement having a weight fraction $\phi_{Al_2O_3}$ of aluminum oxide, the moles of ettringite formed per cubic meter of concrete is

$$C_E = \frac{x_{cem}\phi_{Al_2O_3}}{0.10196} \tag{19}$$

with the Al_2O_3 molar weight of 0.10196 kg per mole.

7.6 Effects of Cracks and Joints

Although not a degradation mechanism *per se*, effects due to the presence of cracks and joints can be incorporated into the models for transport of the ions. Since the roof slab will likely be a supported member such that the bottom of the roof slab will be in tension, it is assumed that cracks will appear on the bottom of the slab and extend upwards to the neutral axis of the slab. In the case of joints, 4SIGHT assumes that the joint extends through the entire depth of the roof slab. The joint is filled with a joint compound with a known permeability and service life.

The permeability of a cracked slab is calculated assuming the crack walls are smooth and parallel. Given a square slab with width L and depth D having cracks with width w penetrating the full depth D of the slab, the permeability of the slab is a weighted sum of

the permeability of the crack, k_c, and the permeability of the uncracked concrete, k_o:

$$k_s = \frac{w}{L+w}k_c + \frac{L}{L+w}k_o \tag{20}$$

$$= \frac{w}{L+w}\frac{w^2}{12} + \frac{L}{L+w}k_o \tag{21}$$

Since $w^3/12$ is typically far greater than Lk_o, the permeability of the slab can be approximated by the permeability due to the crack. Further, if each of the cracks of width w are spaced a distance a apart, the permeability of the slab, k_s, is

$$k_s = \frac{w^3}{12a} \tag{22}$$

Joints can be handled in a similar manner as cracks. However, joints will typically be very much wider than cracks Since joints will presumably extend the entire thickness of the slab, once the joint fails, the flow through the joint would overwhelm the transport of ions through the central portion of the slab. In fact, the transport coefficients could be as great as, or greater than, those of the soil. Therefore, upon failure of the joint, 4SIGHT assumes that the roof fails to impede the flow of water into the vault, the transport properties of the concrete should be approximated by the transport properties of soil, and the calculation ceases.

8 Chemical Equilibrium

After each time step, each computational element is brought to chemical equilibrium by satisfying two conditions:

1. If a salt exists as solid, the constituent ion concentration product equals the solubility product.

2. The sum of the free charges from all available ions equals zero, insuring local charge neutrality.

Given the salt $A_\beta C_\alpha$, composed of anion $A^{\alpha-}$ and cation $C^{\beta+}$, condition 1 above implies that if $A_\beta C_\alpha$ exists as a solid then

$$[A^{\alpha-}]^\beta [C^{\beta+}]^\alpha = K_{sp} \tag{23}$$

where K_{sp} is the ion solubility product. Condition 2 implies that with m anions and n cations present in the pore solution:

$$\sum_{i=1}^{m} \alpha A_i^{\alpha-} = \sum_{j=1}^{n} \beta C_j^{\beta+} \tag{24}$$

4SIGHT satisfies eqns. 23 and 24 using an iterative process. Each iteration is composed of two steps:

1. Determine how many moles of salt should be precipitated or dissolved to satisfy eqn. 23.
2. Given K_{sp} for water, what concentration of hydroxyl ions (OH^-) is needed to satisfy eqn. 24

These two steps are repeated, as necessary, until both equations are satisfied.

9 Failure Criteria

Once the rules for propagating ions through the slab have been defined, criteria for terminating the calculations are needed for each degradation mechanism. Termination occurs when suitable failure criteria have been met.

9.1 Reinforcement Corrosion

Corrosion of the rebar is due to the presence of chloride ions and proceeds in two stages: initiation and corrosion. The initiation stage is the time during which chloride ions are diffusing through the concrete. During this time there is an insufficient concentration of chloride ions at the rebar for corrosion to occur. As the concentration of chloride ions at the rebar increases, the pH decreases to insure charge neutrality. When a sufficient concentration of chloride ions reaches the rebar, corrosion begins. The corrosion stage begins at the onset of corrosion and lasts until failure of the steel reinforcement, which is typically only a few years hence. Since the initiation stage may last for hundreds of years, the duration of the corrosion stage is insignificant to the total lifetime of the reinforcement. Therefore, an accurate estimate of the service life of the concrete can be approximated from the duration of the initiation period.

9.2 Sulfate Attack

As the sulfate front proceeds into the concrete, the effective thickness of the concrete decreases. At some point in time the roof has an insufficient thickness to support its load. At this time, structural failure of the roof will occur. This critical depth of sulfate penetration can only be determined through detailed structural analysis. Therefore, it is the user's responsibility to provide this information.

9.3 Joint Failure

The failure criteria for joints is based upon the service life of the joint compound. Since joint widths will be centimeters wide, as previously stated, upon joint failure the permeability of the roof slab will be overwhelmed by the flow through the joints. Therefore, the calculations terminate upon when the internal time variable has reached the limit of the joint service life.

10 Calculated Material Properties

All of the concrete physical parameters (*e.g.*, diffusivity, permeability, *etc.*) are user specified inputs to `4SIGHT`. However, in cases where not all properties are available, missing quantities must be approximated using existing correlations. The physical properties must be established due to both hydration and leaching. The physical properties due to hydration are the initial conditions. However, as the porosity changes due to leaching, corrected values of the physical parameters are needed.

10.1 Hydration

The hydration of cement can be approximated by a reaction between tri-calcium silicate (C3S) and water, forming a calcium silicate hydrate (CSH). A more elaborate model incorporating multiple mineral phases would require chemical analysis of the cement and yield relatively little additional information concerning degree of hydration. The weight ratio of water to cement, $\frac{w}{c}$, is the oft reported quantity to characterize the concrete mix. After some period of hydration, the fraction of the initial C3S which has hydrated is the degree of hydration, α. The relation between these two properties and porosity can be determined stoichiometrically [7],

$$\phi = 1 - \frac{1 + 1.31\alpha}{1 + 3.2\frac{w}{c}} \tag{25}$$

and is valid for $\frac{w}{c}$ values used in practice.

The diffusivity can be related to either $\frac{w}{c}$ or ϕ. After the first 100 days of hydration the transport properties of most cement pastes are near their asymptotic values. Although the values are still changing after 100 days, these changes are small compared to the accuracy with which these transport measurements can be made. Due to this apparent steady state, an empirical relation between D_O, the chloride diffusivity in m^2/s, and $\frac{w}{c}$ was developed for cement paste by Atkinson, Nickerson, and Valentine[8] and Walton, Plansky, and Smith[9]

$$\log_{10} D_o = 6.0\frac{w}{c} - 9.84 \tag{26}$$

for values of $\frac{w}{c}$ in the range (0.2–0.6).

D_o for cement paste can also be related to ϕ using the universal relation for the formation factor due to hydration, ϑ, [10]:

$$\vartheta = \frac{D_o}{D^f_{Cl^-}} = 0.001 + .07\phi^2 + 1.8(\phi - .18)^2 H(\phi - .18) \tag{27}$$

where D_o is the concrete chloride diffusivity, D_f is the free ion diffusivity of chloride ions, and $H(x)$ is the Heaviside function. The quantity ϑ is a constant for all ions.

The above equations relate the diffusivity of cement paste to water:cement ratio or porosity. A relationship is now needed between cement paste diffusivity and concrete diffusivity. Experimental results of Luping and Nilsson[11] for cement paste and mortar suggest that their diffusivities are approximately equal. Additionally, results from numerical experiments by Garboczi, Schwartz, and Bentz[12] investigating the effects of aggregate-paste interfacial zone diffusivity upon bulk diffusivity suggest that concrete diffusivity is approximately equal to the paste diffusivity.

Given $\frac{w}{c}$, the permeability can be approximated from the data in Hearn, et al.[13]:

$$k = 10^{5.0\frac{w}{c}} \times 10^{-21}\, m^2 \tag{28}$$

for $\frac{w}{c}$ in the range (0.35,0.80).

10.2 Leaching

Once D and k have been established, changes due to leaching can be calculated from $\vartheta(\phi_o)$, where ϕ_o is the initial porosity due to hydration. As the $Ca(OH)_2$ is leached, the porosity increases. Let the value of porosity after leaching be ϕ', and the diffusivity be D'. Unfortunately, the ratio D'/D^f is not simply $\vartheta(\phi')$ because as calcium hydroxide is leached from the paste the ratio D'/D^f does not retrace eqn. 27. Rather, the ratio D'/D^f is greater than $\vartheta(\phi')$, as demonstrated by the NIST microstructural model.

The NIST cement microstructural model[14] was used to determine the relation for D'/D_o upon leaching. Results for a $\frac{w}{c}$=0.35 paste are shown in Figure 2. The formation factor decreases with decreasing porosity due to hydration, denoted by circles. Upon leaching of the calcium hydroxide, the formation factor follows the curve denoted by squares. Given the following definitions:

$$\vartheta_o = \vartheta(\phi_o) \qquad \vartheta' = \vartheta(\phi') \tag{29}$$

An empirical relation was developed to relate the leached pore structure to the undamaged pore structure.

$$\xi = \vartheta_o + 2.0(\vartheta' - \vartheta_o) \tag{30}$$

and is shown by the solid curve in Figure 2. Therefore, the ratio of the leached value of diffusivity, D', to the initial value D_o is

$$\frac{D'}{D_o} = \frac{\xi}{\vartheta_o} \tag{31}$$

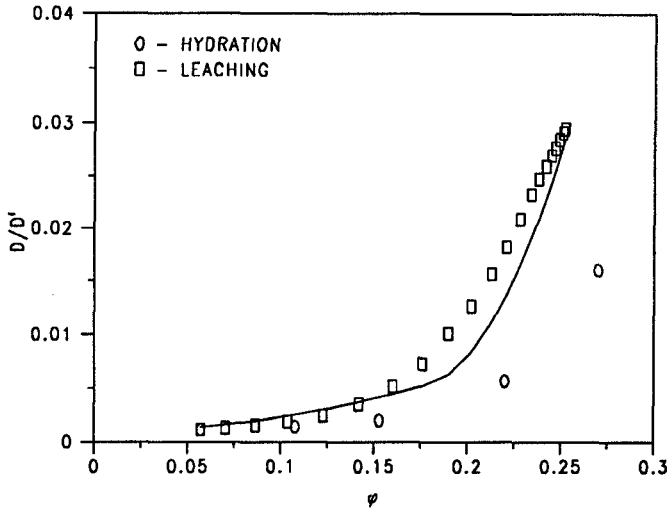

Figure 2: Formation factor, D/D^f, due to hydration (circles) and to leaching (squares) for w/c=0.35 (after [14]). The solid line represents the approximation used by 4SIGHT for the leaching formation factor.

The relative change in permeability can be calculated using ξ. The Katz-Thompson equation relates permeability to ϑ [15]:

$$k = \frac{d_c^2}{226}\vartheta \tag{32}$$

where d_c is the diameter of the largest sphere which can pass through the pore space of the sample. Also, given that $d_c \propto \vartheta$ [16], the relative change in permeability becomes

$$\frac{k'}{k_o} = \left(\frac{\xi}{\vartheta_o}\right)^3 \tag{33}$$

The final dimensionless advection-diffusion equation is a combination of eqns. 14, 31, and 33:

$$\frac{\partial c_i}{\partial T} = \nabla \cdot \frac{\xi_i}{\vartheta_o}\nabla P + \left(\frac{\xi_i}{\vartheta_o}\right)^3 \mathbf{v}_D \cdot \nabla c_i \tag{34}$$

There is one such equation for chemical species i.

11 Program Outline

The flow of the program is summarized by the pseudo-code program shown in Figure 3. The program naturally divides itself into three parts: initialization, ion propagation, and final state output.

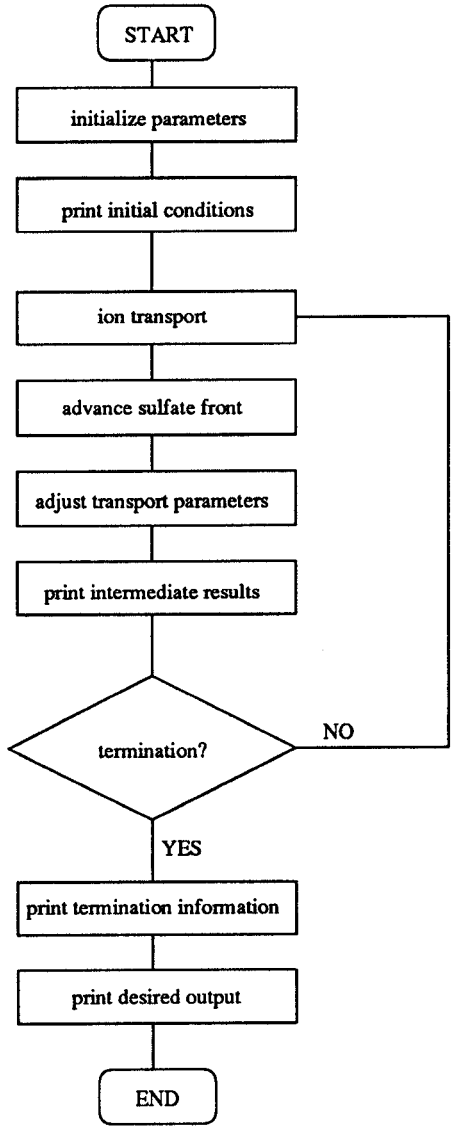

Figure 3: 4SIGHT flow chart.

12 Assumptions

The calculations of 4SIGHT are based upon a set of assumptions. For completeness, these are assumptions are enumerated here:

1. The vault system is, and remains, water saturated. This should be valid for all but the most arid locations.

2. Darcy flow is valid, even at the bottom of the slab which might not be in direct contact with liquid.

3. There is sufficient oxygen present for corrosion. This a conservative approach to corrosion.

4. The concrete vault begins service approximately 100 days after casting. Therefore, the transport properties have reached about 90% of their assymptotic values.

5. The external ion concentration and hydraulic head are constant over the life of the vault.

6. As the sulfate front advances, the concrete in its wake can be treated like soil and the external conditions can be advanced to this point.

13 Acknowledgements

The authors wish to acknowledge the financial support of the U.S. Nuclear Regulatory Commission (NRC). Mr. Jacob Philip was the NRC Program Manager and his advice was greatly appreciated. The authors would also like to express their thanks to Dr. James Pommersheim of Bucknell University and Mr. Dale Bentz of NIST for their discussion and comments.

References

[1] J.R. Clifton, and L.I. Knab, "Service Life of Concrete," *NISTIR 89-4086*, U.S. Department of Commerce, (1989).

[2] J.M. Pommersheim, and J.R. Clifton, "Models of Transport Processes in Concrete," *NISTIR 4405*, U.S. Department of Commerce, (1990).

[3] R.B. Bird, W.E. Stewart, and E.N. Lightfoot, *Transport Phenomenon*, John Wiley & Sons, New York, (1960).

[4] J.C. Slattery, *Mass Transport*, McGraw-Hill Book Company, New York, (1972).

[5] A. Atkinson, and J.A. Hearne, "Mechanistic Models For The Durability Of Concrete Barriers Exposed To Sulfate-Bearing Groundwaters," *MRS Proc.*, **176**, 149–156, (1990).

[6] H.F.W. Taylor, *Cement Chemistry*, Academic Press, New York, (1990).

[7] Young and Hansen, "Volume Relationships for C-S-H Formation Based on Hydration Stoichiometries," *Proc. MRS*, Eds. Struble and Brown, **85**, 313–322, (1987).

[8] A. Atkinson, A.K. Nickerson, and T.M. Valentine, "The Mechanism of Leaching From Some Cement-Based Nuclear Wasteforms," *Radioactive Waste Management and the Nuclear Fuel Cycle*, **4** (4), 357–378, (1984).

[9] J. Walton, L. Plansky, and R. Smith, "Models for Estimation of Service Life of Concrete Barriers in Low-Level Radioactive Waste Disposal," NUREG/CR-5542, EGG-2597, (1990).

[10] E.J. Garboczi and D.P. Bentz, "Computer simulation of the diffusivity of cement-based materials," *J. Materials Science*, **27**, 2083–2092, (1992).

[11] T. Luping, and L. Nilsson, "Rapid Determination of the Chloride Diffusivity in Concrete by Applying an Electric Field," *ACI Materials Journal*, **89** (1), 49–53, (1992).

[12] E.J. Garboczi, L.M. Schwartz, and D.P. Bentz, "Modelling the influence of the interfacial zone on the D.C. electrical conductivity of mortar," (Submitted to) *Journal of the ACBM*.

[13] N. Hearn, R.D. Hooton, and R.H. Mills, "Pore Structure and Permeability," *Significance of Tests and Properties of Concrete and Concrete-Making Materials*, Eds. Klieger and Lamond, ASTM, 240–262, (1994).

[14] D.P. Bentz and E.J. Garboczi, "Guide to Using HYDRA3D: A Three-Dimensional Digital-Image-Based Cement Microstructural Model," NIST-IR 4746, U.S. Department of Commerce, (1992).

[15] A.J. Katz, and A.H. Thompson, "Quantitative Prediction of Permeability in Porous Rock," *Phys. Rev. B*, **34** (11), 8179–8181, (1986).

[16] S. Li, and D.M. Roy, "Investigation Of Relations Between Porosity, Pore Structure, and Cl^- Diffusivity Of Fly Ash and Blended Cement Pastes," *Cement and Concrete Research*, **16** (5), 749, (1986).

[17] T. Mason and D. Brown, *Lex & Yacc*, O'Reilly & Associates, Inc., Sebastopol, CA, (1991).

Appendix A

A Input Parsing

The input parsing was accomplished using public domain versions of LEX and YACC written for the PC. These tools simplify parsing user input, facilitating modification and extensions in program capabilities. The user unfamiliar with LEX and YACC should read the book by Mason and Brown listed in the references. To obtain a copy of the public domain version, access the following directory through anonymous ftp:

machine: `omnigate.clarkson.edu`
directory: `pub/msdos/djgpp`

LEX and YACC are programs that take as input user specifications for a particular syntax and create a C file which performs the functions specified in the user LEX and YACC files. The resulting C file is then simply `#include`-ed into the main C program. Of course, the user can code the parsing directly in C, but LEX and YACC make the coding less errorsome and more extensible.

A.1 LEX Specification

```
%{
#include "4sight_t.h"
%}
integer     [0-9]+
dreal       ([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)
ereal       ({dreal}|{integer})[eE][+-]?[0-9]+
real        {dreal}|{ereal}
diff        [Dd][Ii][Ff][Ff]
perm        [Pp][Ee][Rr][Mm]
wc          [Ww][Cc]
thick       [Tt][Hh][Ii][Cc][Kk][Nn][Ee][Ss][Ss]
neuax       [Nn][Aa][Xx][Ii][Ss]
youngs      [Yy][Oo][Uu][Nn][Gg][Ss]
beta        [Bb][Ee][Tt][Aa]
ce          [Cc][Ee]
rough       [Rr][Oo][Uu][Gg][Hh][Nn][Ee][Ss][Ss]
gamma       [Gg][Aa][Mm][Mm][Aa]
poisson     [Pp][Oo][Ii][Ss][Ss][Oo][Nn]
extrn       [Ee][Xx][Tt][Ee][Rr][Nn][Aa][Ll]
intrn       [Ii][Nn][Tt][Ee][Rr][Nn][Aa][Ll]
time        [Tt][Ii][Mm][Ee]
depth       [Dd][Ee][Pp][Tt][Hh]
rebar       [Rr][Ee][Bb][Aa][Rr]
output      [Oo][Uu][Tt][Pp][Uu][Tt]
head        [Hh][Ee][Aa][Dd]
crack       [Cc][Rr][Aa][Cc][Kk]
joint       [Jj][Oo][Ii][Nn][Tt]
at          [Aa][Tt]
until       [Uu][Nn][Tt][Ii][Ll]
assign      "="|":"
wspace      [ \t]+
kill        (([Qq]([Uu][Ii][Tt])?)|([Ee]([Xx][Ii][Tt])?))
nl          \n
%%
{wspace}        ;
OH              { return HYDROXIDE;}
Cl              { return CHLORINE;}
SO4             { return SULFATE;}
```

```
CO3             { return CARBONATE;}
H               { return HYDROGEN;}
Ca              { return CALCIUM;}
Na              { return SODIUM;}
K               { return POTASSIUM;}
{diff}          { return DIFF;}
{perm}          { return PERM;}
{wc}            { return WC;}
{thick}         { return THICKNESS;}
{neuax}         { return NEUAX;}
{youngs}        { return YOUNGS;}
{beta}          { return BETA;}
{ce}            { return Ce;}
{rough}         { return ROUGHNESS;}
{gamma}         { return GAMMA;}
{poisson}       { return POISSON;}
{extrn}         { return EXTRN;}
{intrn}         { return INTRN;}
{time}          { return TIME;}
{depth}         { return DEPTH;}
{rebar}         { return REBAR;}
{output}        { return OUTPUT;}
{head}          { return HEAD;}
{crack}         { return CRACK;}
{joint}         { return JOINT;}
{at}            { return AT;}
{until}         { return UNTIL;}
{integer}       { sscanf(yytext,"%ld", &yylval.int32);
                      return INT32;}
{real}          { sscanf(yytext,"%lf", &yylval.real);
                      return REAL;}
{assign}        { return EQUALS;}
{nl}            { return NEWLINE;}
{kill}          { return KILL;}
.               { return yytext[0];}
%%
```

A.2 YACC Specification

```
%{
int yylex(void);
void yyerror(char *);
void yyerror(s)
   char *s;
{
 /* fprintf(stderr,"%s \t see line %d\n",s,input_file_line); */
}

void syntax_error(char *);
void syntax_error(char *s)
{
   fprintf(stderr,"%s \t see line %d\n",s,input_file_line);
}

%}

%union {
        double  real;                   /* real value */
        long    int32;                  /* DoubleWord Integer */
        int     int16;                  /* SingleWord Integer */
}

%token <real> REAL
%token <int32> INT32
%token KILL
%token EQUALS
%token NEWLINE
%token DIFF
%token PERM
%token WC
%token THICKNESS
%token NEUAX
%token YOUNGS
%token BETA
%token Ce
%token ROUGHNESS
%token GAMMA
```

```
%token POISSON
%token EXTRN
%token INTRN
%token HYDROGEN
%token CALCIUM
%token SODIUM
%token POTASSIUM
%token HYDROXIDE
%token CHLORINE
%token SULFATE
%token CARBONATE
%token TIME
%token DEPTH
%token REBAR
%token OUTPUT
%token HEAD
%token CRACK
%token JOINT
%token AT
%token UNTIL


%type <real> expr
%type <int16> anion
%type <int16> cation

%left '+' '-'
%left '*' '/'

%%

lines:                  /* NOTHING */
        | lines line
        ;

line:     NEWLINE       {input_file_line++;}
        | terminate     { printf("%d lines parsed.\n",input_file_line);
                          return 0;}
        | assign
        | recvr NEWLINE
```

```
terminate:        KILL      {printf("Received KILL.\n");}
        ;

recvr:     error            { syntax_error("Syntax error");}
        | recvr error
        | recvr EQUALS expr { syntax_error("Unknown Command");}
        ;

expr:      INT32            {$$ = (real)$1;}
        | REAL              {$$ = $1;}
        ;

assign:    DIFF EQUALS expr       { Dinfty.value = $3;
                                    Dinfty.is_default = FALSE;}
        | PERM EQUALS expr        { kinfty.value = $3;
                                    kinfty.is_default = FALSE;}
        | WC EQUALS expr          { wc.value = $3;
                                    wc.is_default = FALSE;}
        | THICKNESS EQUALS expr   { thickness.value = $3;
                                    thickness.is_default = FALSE;
                                    sample_length = thickness.value;}
        | NEUAX EQUALS expr       { neutral_axis_depth.value = $3;
                                    neutral_axis_depth.is_default=FALSE;}
        | YOUNGS EQUALS expr      { Youngs.value = $3*1.0E+09;
                                    Youngs.is_default = FALSE;}
        | BETA EQUALS expr        { beta.value = $3;
                                    beta.is_default = FALSE;}
        | Ce EQUALS expr          { CE.value = $3;
                                    CE.is_default = FALSE;}
        | ROUGHNESS EQUALS expr   { roughness.value = $3;
                                    roughness.is_default = FALSE;}
        | GAMMA EQUALS expr       { gamma.value = $3;
                                    gamma.is_default = FALSE;}
        | POISSON EQUALS expr     { nu.value = $3;
                                    nu.is_default = FALSE;}
        | EXTRN cation EQUALS expr  { cation[$2].c[0] = $4;}
        | EXTRN anion  EQUALS expr  { anion[$2].c[0] = $4;}
        | INTRN cation EQUALS expr  { cation[$2].c[1] = $4;}
```

```
        | INTRN anion  EQUALS expr  { anion[$2].c[1] = $4;}
        | TIME EQUALS expr          { termination_type = TIME_LIMIT;
                                      if(MaxDay.is_default==TRUE)
                                       MaxDay.value = $3;
                                      else
                                       MaxDay.value = MIN(MaxDay.value,$3);
                                       MaxDay.is_default = FALSE;}
        | DEPTH EQUALS expr         { termination_type = STRUCT_LIMIT;
                                      MaxDepth = $3;}
        | REBAR EQUALS expr         { rebar_depth=$3;
                                      termination_type = STRUCT_LIMIT;}
        | OUTPUT cation anion       { sol_array[$2][$3].output_flag=TRUE;}
        | OUTPUT cation             { cation[$2].output_flag = TRUE;}
        | OUTPUT anion              { anion[$2].output_flag = TRUE;}
        | HEAD EQUALS expr          { head.value = $3;
                                      head.is_default = FALSE;}
        | CRACK EQUALS expr AT expr DEPTH expr
                                      { crack_width.value = $3;
                                        crack_width.is_default = FALSE;
                                        crack_spacing = $5;
                                        crack_depth = $7;}
        | JOINT EQUALS expr AT expr UNTIL expr
                        { joint_width = $3;
                          joint_spacing = $5;
                          joint_lifetime = $7*365;
                          if(MaxDay.is_default == TRUE)
                           MaxDay.value = joint_lifetime;
                          else
                           MaxDay.value = MIN(MaxDay.value,joint_lifetime);
                          MaxDay.is_default = FALSE;
                          joint_is_specified = TRUE;}
        | JOINT PERM EQUALS expr        { joint_permeability = $4;}
        ;

cation:   HYDROGEN              { $$ = H;}
        | CALCIUM               { $$ = Ca;}
        | SODIUM                { $$ = Na;}
        | POTASSIUM             { $$ = K;}
        ;
```

```
anion:    HYDROXIDE              { $$ = OH;}
        | CHLORINE               { $$ = Cl;}
        | SULFATE                { $$ = SO4;}
        | CARBONATE              { $$ = CO3;}
        ;
%%
```

B CWEB - Source Code

The source code (except for the parsing described above) was written using the literate programming tool CWEB written by Donald Knuth and Silvio Levy. The CWEB tool is composed of two programs: `cweave` and `ctangle`. The user writes a CWEB file using any ASCII editor. This file contains both the C code and the documentation. `ctangle` takes this input file and extracts the C code. Similarly, `cweave` takes the input file and creates a TEXfile which can be TEX-ed, creating a formatted version of the complete documentation, including the source code. Therefore, to use these tools, the user must have both CWEB and TEXfor the PC. Fortunately, both of these tools are public domain, and can be accessed by anonymous ftp:

TEX	machine:	`ftp.njit.edu`
	directory:	`pub/msdos/emtex`
CWEB	machine:	`labrea.stanford.edu`
	directory:	`pub/cweb`

The only modification made to CWEB for the development of 4SIGHT was the extension to LTEX. To use the extensions of LTEXwith CWEB, one only needs a LTEXstyle sheet. This too is available by anonymous ftp:

machine:	`ftp.th-darmstadt.de`
directory:	`pub/programming/literate-programming/c.c++`

The pages that follow are verbatim output from `cweave`, via LTEX:

LLW 4SIGHT Analysis

(Version 1.0)

5 JUN 1995

Contents

1. INTRODUCTION.

The 4SIGHT computer program has been developed by the Inorganic Group of the Building Materials Division at NIST. This program facilitates estimating the service life of low level nuclear waste storage sites by determining the hydraulic conductivity of the vault roof as a function of time. This objective is acheived by considering prominent deterioration mechanisms to predict the intrinsic permeability, effluent, and working thickness of the vault slab.

The condition of the vault slab is assesssed by modelling the ingress of deleterious ion species by both diffusion and convective flow due to hydraulic pressure gradients. Additionally, changes in physical properties due to the leaching/deposition of salts are incorporated into the program. The propagation of ions are calculated using an advection-diffusion equation. After each discrete time step, each computational cell is brought to chemical equilibrium through precipitation/dissolution of ionic salts.

The program incorporates a one-dimensional model of the vault since the critical deterioration will most likely occur at the ceiling of the vault. The ceiling is the most likely element to experience moisture, stress, and external ions such as Cl^- and SO_4. Addtionally, the majority of the transport will occur at the center of the roof slab since the edges of the slab will be supported. Therefore, a simple one-dimension model of the flow at the center of the slab will give a conservative estimate of the transport for the entire slab.

The program proceeds in a strainghtforward manner; The program gets input parameters from the user; initializes the ion information and scaling parameters; iterates diffusion and convection calculations, bringing each computational element to chemical equilibrium between iterations; determines whether failure has occured; and prints pertenent information about the performance of the vault.

The output of the program can be printed to either the screen or to an ASCII file. The ASCII file can then be used as input to a spreadsheet program for further analysis.

2. MAIN.

The body of 4SIGHT is fairly straightforward and the logical flow can be inferred from the outline below.

```
#define banner "\"\tThis␣is␣4SIGHT␣(Version␣1.0)\"\n\n"
#define NUM_CELLS 20
#define NUM_SURFACES (NUM_CELLS + 1)
#define DLIMIT 0.0005
  ⟨Include files 3⟩;
  ⟨Preprocessor definitions 11⟩;
  ⟨User defined data types 14⟩;
  ⟨Global variables 4⟩;
  ⟨Function declarations 23⟩;
  ⟨Interrupt handlers 24⟩;
  ⟨Input parsing routine 13⟩;
  int main(int argc, char **argv)
  {
    int i, j, k;
    ⟨System setup 5⟩;
    ⟨Parse input 6⟩;
    ⟨Parameter initialization 7⟩;
    ⟨Print header 75⟩;
    ⟨Print intermediate results 76⟩;
    do {
      if (kbhit()) {
        i = getch();
        control_break();
      }
      ⟨Ion transport 8⟩;
      ⟨Adjust parameters 9⟩;
      ⟨Print intermediate results 76⟩;
      ⟨Assess termination 10⟩;
    } while (¬termination);
    ⟨Print termination information 77⟩;
    ⟨Print desired output 78⟩;
    return 1;
  }
```

3. The following are `#include` files needed for some of the intrinsic functions such as *printf* and *scanf*. The math functions *pow* and *log* and the math error routine *matherr* require `math.h`. Additional `include` files are declared when needed. The routine *kbhit*( ) requires the header file `conio.h`.

```
⟨Include files 3⟩ ≡
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <bios.h>
#include <time.h>
```

See also sections 22, 28, and 32.

This code is used in section 2.

4. ⟨ Global variables 4 ⟩ ≡

```
long elapsed_time;
```

See also sections 12, 19, 20, 21, 29, 31, 40, 63, and 69.
This code is used in section 2.

5. System Setup.

```
#define CTRL_BRK_ON 1
#define GET_TIME 0
```

⟨ System setup 5 ⟩ ≡

```
printf(banner);
elapsed_time = biostime(GET_TIME, 0L);      /* get BIOS timer value */
⟨ Determine STDIN 30 ⟩;
i = setcbrk(CTRL_BRK_ON);      /* check ctrl-brk every system call */
ctrlbrk(control_break);
signal(SIGFPE, div_0);
initialize_ion_data( );
```

This code is used in section 2.

6. Parse Input.

⟨ Parse input 6 ⟩ ≡

```
i = yyparse( );
```

This code is used in section 2.

7. Parameter Initialization.

⟨ Parameter initialization 7 ⟩ ≡

```
initialize_parameters( );
update_pressures( );
aci_guidlines( );
chemical_equilibrium(FALSE);
chemical_equilibrium(FALSE);
chemical_equilibrium(FALSE);
⟨ Print initial system state 74 ⟩;
```

This code is used in section 2.

8. Ion Transport.

⟨ Ion transport 8 ⟩ ≡

```
ion_diffusion( );
chemical_equilibrium(TRUE);
advance_sulfate_front( );
```

This code is used in section 2.

9. Adjust Parameters.

⟨ Adjust parameters 9 ⟩ ≡

```
⟨ Advance global clock 70 ⟩;
⟨ Adjust physical parameters 68 ⟩;
update_pressures( );
⟨ Update the porosities 67 ⟩;
```

This code is used in section 2.

10. Assess Termination.

```
⟨ Assess termination 10 ⟩ ≡
  ⟨ Assess chloride penetration 64 ⟩;
  ⟨ Assess simulation termination 72 ⟩;
  ⟨ Check time dependencies 71 ⟩;
```

This code is used in section 2.

11. The following are useful macro definitions which save a lot of typing.

```
⟨ Preprocessor definitions 11 ⟩ ≡
#define SQR(a)  ((a) * (a))
#define CUB(a)  ((a) * (a) * (a))
#define MAX(a, b)  ((a > b) ? a : b)
#define MIN(a, b)  ((a < b) ? a : b)
```

See also section 46.

This code is used in section 2.

12. Declare *yyparse* which is written in YACC and LEX. The function *yyparse* takes no arguments and simply parses the input, assigning values to global variables. The routine does not "return" until the entire input stream has been parsed.

```
⟨ Global variables 4 ⟩ +≡
  int input_file_line = 1;      /* line number of input file (lex/yacc) */
```

13.

```
⟨ Input parsing routine 13 ⟩ ≡
  int yyparse(void);
#include "yyparse.c"
```

This code is used in section 2.

14. User defined data structures.

These are constructed datatypes used by 4SIGHT. Two fundamental types are **real** and **boolean**. Using **real** lets the programmer easily change between **float** and **double** data types for real numbers.

The boolean types are defined {FALSE,TRUE} because Borland assigns the first entry the number 0. Variables defined as **boolean** can be used in a conditional statement as **if**(*myboolvar*) without direct comparison to FALSE or TRUE.

```
⟨ User defined data types 14 ⟩ ≡
  typedef double real;
  typedef enum {
    FALSE, TRUE
  } boolean;
```

See also sections 15, 16, and 17.

This code is used in section 2.

15. The ION data type contains all important information about a single ion. This information includes:

c[]: concentration at each computational cell surface.

η: the ratio D_{fi}/D_o for each ion.

D_f: free ion diffusivity.

valence: the valence of the ion.

output_flag: a flag to denote whether the final ion concentration, as a function of distance, is printed in system.out.

name[]: the ASCII name of the ion.

```
⟨ User defined data types 14 ⟩ +≡
  typedef struct {
    real c[NUM_SURFACES];     /* concentration at each surface */
    real moles[NUM_SURFACES];     /* moles ion in solution */
    real η;     /* Dfi/Do */
    real Df;     /* ion free diffusivity */
    real valence;     /* ion valence */
    boolean output_flag;     /* print to system.out? */
    char name[5];     /* ASCII name of ion */
  } ION;
```

16. The SOLARRAY data type contains the solubility products and stiochiometric ratios for each anion-cation combination. The salt is C_mA_n, where C represents the cation, and A the anion. The quantity of solid is defined on the cell surfaces.

```
⟨ User defined data types 14 ⟩ +≡
  typedef struct {
    real s[NUM_SURFACES];     /* moles solid salt in element */
    real ksp;     /* salt solubility constant */
    real m;     /* stiochiometric cation factor */
    real n;     /* stiochiometric anion factor */
    real molar_density;     /* cm^3/mole */
    boolean output_flag;     /* print salt to system.out? */
  } SOLARRAY;
```

17. The MATERIAL datatype is used for any material property that can be specified by the user. It carries a flag to denote whether the value is the default one, or the one specified by the user.

```
⟨ User defined data types 14 ⟩ +≡
  typedef struct {
    real value;
    boolean is_default;
  } MATERIAL;
```

18. Global variables.

Most of these are variables that will be changed by the input file. The listing is broken into major groups: material parameters, scaling parameters, etc.

19. Transport parameters. These are the parameters defined at each element that determine the rate of transport. These include the inverse formation factor ξ, pressure Ψ, and the porosity ϕ.

```
⟨ Global variables 4 ⟩ +≡
  real ξ[NUM_CELLS];      /* changing formation factor */
  real ξ'[NUM_SURFACES];      /* changing formation factor at the surface */
  real Ψ[NUM_SURFACES];      /* Pressure */
  real φ^n[NUM_CELLS];      /* current porosity */
  real φ'^n[NUM_SURFACES];      /* porosity at the surface */
  real φ'^{n-1}[NUM_SURFACES];      /* previous porosity at concentration */
  real litre[NUM_SURFACES];      /* litres of solution (surface area)*/
  real strength[NUM_SURFACES];      /* strength in psi */
  int FIRST_CELL = 0;      /* first active cell number */
```

20. User specified parameters. These parameters determine the ultimate properties of the sample. Although default values are given, user specified values will override these values.

```
⟨ Global variables 4 ⟩ +≡
  MATERIAL thickness = {1.0, TRUE};      /* roof thickness */
  MATERIAL neutral_axis_depth = {0.75, TRUE};      /* depth of neutral axis */
  MATERIAL k_∞ = {2.5 · 10^-18, TRUE};      /* ultimate permeability */
  MATERIAL D_∞ = {5.7 · 10^-12, TRUE};      /* ultimate diffusivity */
  MATERIAL head = {5.0, TRUE};      /* external pressure head */
  MATERIAL wc = {0.45, TRUE};      /* water:cement */
  MATERIAL Youngs = {20.0 · 10^+09, TRUE};      /* Young's modulus */
  MATERIAL β = {1.8 · 10^-06, TRUE};      /* linear strain coefficient */
  MATERIAL CE = {350., TRUE};      /* conc. sulfate as ettringite */
  MATERIAL roughness = {1.0, TRUE};      /* roughness factor */
  MATERIAL γ = {10.0, TRUE};      /* fracture surface energy */
  MATERIAL ν = {0.2, TRUE};      /* Poisson ration */
```

21. Scaling parameters.

```
⟨ Global variables 4 ⟩ +≡
  real L = 0.010;      /* length of ELEMENTS so that ΔX = 1 */
  real D_o = 1.0 · 10^-11;      /* scaling diffusivity */
  real k_o = 1.0 · 10^-18;      /* scaling permeability */
  real ΔT = 1.0;      /* dimensionless time increment */
  real ΔX[NUM_SURFACES];      /* dimensionless length between each cell */
```

22. Interrupt handlers.

The following routines handle *ctrl-brk* and math exceptions. The routine *ctrlbrk* requires `dos.h`. The routine *signal* requires `signal.h`.

⟨ Include files 3 ⟩ +≡

```
#include <dos.h>
#include <signal.h>
```

23. ⟨ Function declarations 23 ⟩ ≡

```
int control_break(void);
int matherr(struct exception *);
void div_0(void);
```

See also sections 33, 39, 43, 51, 54, 56, 59, 60, 61, and 65.

This code is used in section 2.

24. *control_break* handles interruptions due to the use striking `CTRL-BRK` during execution.

⟨ Interrupt handlers 24 ⟩ ≡

```
int control_break(void) {
    int i, j, k;
    printf("\n\n\tUSER␣CTRL-BRK!.\n\n"); ⟨ Print desired output 78 ⟩
    return 0; }
```

See also sections 25 and 26.

This code is used in section 2.

25. The routine *matherr* handles exceptions from the math coprocessor.

⟨ Interrupt handlers 24 ⟩ +≡

```
int matherr(struct exception *e)
{
  printf("\n\n");
  printf("\tmatherr:␣");
  switch (e->type) {
  case DOMAIN: printf("DOMAIN");
    break;
  case SING: printf("SINGULARITY");
    break;
  case OVERFLOW: printf("OVERFLOW");
    break;
  case UNDERFLOW: printf("UNDERFLOW");
    break;
  case TLOSS: printf("TLOSS");
    break;
  default: printf("UNKNOWN␣ERROR␣TYPE!");
    break;
  }
  printf("\n");
  printf("\n\n\t");
  printf("function:\t%s(%lf,%lf)\n", e->name, e->arg1, e->arg2);
  printf("\n");
  exit(0);
}
```

26. The routine *div_0* handles possible divide by zero errors.

```
⟨ Interrupt handlers 24 ⟩ +≡
  void div_0(void)
  {
    int i, j, k;
    printf("\n\n\tPossible␣divide␣by␣0!\n\n");
    ⟨ Print desired output 78 ⟩
    exit(1);
  }
```

27. INITIALIZATION.

28. Redirection of *stdin*. Allow the user to either use redirection at the command line or to specify the input file directly. If an input file is given at the command line without redirection (<), re-define *stdin* to come from the input file. The **#include** files (`fcntl.h` and `sys/stat.h`) are for the routine *dup2*( ), and `errno.h` is required for the global variable *errno*.

```
#define STDIN 0
⟨ Include files 3 ⟩ +≡
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
```

29. If redirection is not used, set *input_file_handle* to the file specified at the command line.

```
⟨ Global variables 4 ⟩ +≡
  int input_file_handle;      /* file containing input parameters */
```

30. If an input file was given without redirection, use *dup2*( ) to copy the input file handle number to the existing *stdin* file handle.

```
⟨ Determine STDIN 30 ⟩ ≡
  if (argc > 1) {      /* redirect from input file */
    fprintf(stderr, "Calculating...");
    if ((input_file_handle = open(argv[1], O_RDONLY)) ≡ -1) {
      switch (errno) {
      case ENOENT: fprintf(stderr, "No such file: %s\n", argv[1]);
        break;
      case EMFILE: fprintf(stderr, "Too many open files.\n");
        break;
      case EACCES: fprintf(stderr, "Permission denied.\n");
        break;
      case EINVACC: fprintf(stderr, "Invalid access code.\n");
        break;
      default: fprintf(stderr, "Error: Unknown error code.\n");
        break;
      }
      exit(0);
    }
    dup2(input_file_handle, STDIN);
  }
  else      /* interactive mode */
    printf("Enter commands:\n");
```

This code is used in section 5.

31. Initialize solubility array.

sol_array is vital to determining the chemical equilibrium of each element. This procedure also initializes the two ION arrays: *anion* and *cation*

```
⟨ Global variables 4 ⟩ +≡
  int num_cations;
  int num_anions;
  SOLARRAY **sol_array;
  ION *cation;
  ION *anion;
```

32. The routine *malloc*( ) requires `stdlib.h`.

```
⟨ Include files 3 ⟩ +≡
#include <stdlib.h>
```

33.

```
⟨ Function declarations 23 ⟩ +≡
  void initialize_ion_data(void);
```

34. Initializing ion and salt array. The data for the ions and salts is in the file `ion.db`. Must first determine the number of anions and cations, including OH^- and H^+, and then allocate memory for *anion*, *cation*, and *sol_array*.

```
void initialize_ion_data( )
{
  int i, j, k;
  FILE *ion_database;      /* ion and salt data base */
  if ((ion_database = fopen("ion.db","rt")) ≡ Λ) {
    printf("\n\tERROR:␣␣Unable␣to␣open␣ion.db!\n\n");
    exit(0);
  }
  fscanf(ion_database,"%d␣%d",&num_cations,&num_anions);
  ⟨ Allocate ion and salt arrays 35 ⟩
  ⟨ Input ion data 36 ⟩
  ⟨ Input salt data 37 ⟩
  fclose(ion_database);
  ⟨ Zero-out ion and salt data 38 ⟩
}
```

35. Allocate memory for all of the arrays. The total number of cations/anions includes OH^- and H^+.

```
⟨ Allocate ion and salt arrays 35 ⟩ ≡
  anion = (ION *) malloc(num_anions * sizeof(ION));
  cation = (ION *) malloc(num_cations * sizeof(ION));      /* allocate 2-D array */
  sol_array = (SOLARRAY **) malloc(num_cations * sizeof(SOLARRAY *));
  for (i = 0; i < num_cations; i++) {
    sol_array[i] = (SOLARRAY *) malloc(num_anions * sizeof(SOLARRAY));
  }
```

This code is used in section 34.

36. Read in the ion data from *ion_database*, cations first, anions second.

```
⟨ Input ion data 36 ⟩ ≡
  for (i = 0; i < num_cations; i++) {
    fscanf(ion_database, "%s", cation[i].name);
    fscanf(ion_database, "%lf", &cation[i].valence);
    fscanf(ion_database, "%le", &cation[i].D_f);
    cation[i].output_flag = FALSE;
  }
  for (j = 0; j < num_anions; j++) {
    fscanf(ion_database, "%s", anion[j].name);
    fscanf(ion_database, "%lf", &anion[j].valence);
    fscanf(ion_database, "%le", &anion[j].D_f);
    anion[j].output_flag = FALSE;
  }
```

This code is used in section 34.

37. Read salt data in cation major order.

```
⟨ Input salt data 37 ⟩ ≡
  for (i = 0; i < num_cations; i++)
    for (j = 0; j < num_anions; j++) {
      fscanf(ion_database, "%le", &sol_array[i][j].ksp);
      fscanf(ion_database, "%lf", &sol_array[i][j].m);
      fscanf(ion_database, "%lf", &sol_array[i][j].n);
      fscanf(ion_database, "%lf", &sol_array[i][j].molar_density);
      sol_array[i][j].output_flag = FALSE;
    }
```

This code is used in section 34.

38. Initialize the ion and salt data to zeros.

```
⟨ Zero-out ion and salt data 38 ⟩ ≡
  for (k = 0; k < NUM_SURFACES; k++) {
    for (i = 0; i < num_cations; i++) {
      cation[i].c[k] = 0.0;
      cation[i].moles[k] = 0.0;
      for (j = 0; j < num_anions; j++) {
        anion[j].c[k] = 0.0;
        anion[j].moles[k] = 0.0;
        sol_array[i][j].s[k] = 0.0;
      }
    }
  }
```

This code is used in section 34.

39. Initialize parameters.

The scale and physical parameters must be set based upon the information from the parsing and ion initialization routines.

```
#define OH 0
#define Cl 1
#define SO4 2
#define CO3 3
#define H 0
#define Ca 1
#define Na 2
#define K 3
#define TIME_LIMIT 0
#define STRUCT_LIMIT 1
#define PRINT_OUTS 20.0
⟨ Function declarations 23 ⟩ +≡
  void initialize_parameters( );
```

```
40. ⟨ Global variables 4 ⟩ +≡
  real sample_length = 1.0;      /* length of sample in meters */
  real sulfate_failure_year;
  real chloride_failure_year;
  real κ[NUM_CELLS];      /* perm. coefficient due to cracking */
  real sulfate_rate;      /* dimensionless rate */
  real sulfate_rate_ms;      /* m/sec */
  real sulfate_depth = 0.0;      /* depth of sulfate penetration */
  real chloride_depth = 0.0;      /* depth of chloride penetration */
  real rebar_depth = 1.0;
  boolean termination = FALSE;      /* simulation termination */
  int termination_type = TIME_LIMIT;      /* limiting condition for termination */
  MATERIAL MaxDay = {10000.0, TRUE};      /* time limit (day) */
  real MaxDepth = 100.0;      /* penetration depth limit (m) */
  MATERIAL crack_width = {0.000100, TRUE};
  real crack_spacing = 100.0;
  real crack_depth = 0.0;
  real crack_permeability;
  real joint_width = 0.0;
  real joint_spacing = 100.0;
  real joint_lifetime = 20000.0;
  boolean joint_change_flag = FALSE;
  boolean joint_is_specified = FALSE;
  real joint_permeability = 0.0;
  real Pout;      /* dimensionless ext. pressure */
  real dP;      /* dimensionless rho g h pressure */
  real υ[NUM_CELLS];      /* Darcy velocity */
  real μ = 0.001;      /* pore fluid viscosity */
  real g = 9.8;      /* gravitational constant(m/sec/sec) */
  real ρ = 1000;      /* density of pore fluid (kg/m^3) */
  real k_B;      /* bulk permeability */
  real D_B;      /* bulk diffusivity */
```

```
real φ = 0.16;      /* porosity */
real F;      /* formation factor */
real α;      /* degree of hydration */
real f;      /* initial solids fraction */
real Vsample;      /* volume of element */
real ρ_CH = 33.1;      /* density of CH */
real molesCH = 13.0;      /* moles CH/ltr soln. */
real ϑ_0 = 0.00281;      /* initial 1/F */
real ϑ;      /* current 1/F */
real print_day;      /* next day to print interm. results */
real print_day_interval;      /* days between interm. results */
real earliest_failure_day;      /* earliest structural failure */
```

41. *initialize_parameters*( ). This routine initializes the scaling parameters and physical parameters.

```
void initialize_parameters( )
{
  int i, j, k;
  int n, m;
  int iter;
  real moles_cation, moles_anion, charge, x, dx, z, critical_Cl_concentration, T_crit,
      concen_ratio, darcy_velocity, perm_time, chloride_time, perm_factor,
    /* modify DT to account for high permeability */
  perm_depth, A, crack_factor;      /* modify DT to account for cracks */
  for (k = FIRST_CELL; k < NUM_SURFACES; k++)
    ΔX[k] = 1.0;
  ⟨ Establish material parameters of concrete 42 ⟩
  aci_211( );
  ⟨ Propagate ion concentration in concrete 45 ⟩
  ⟨ Print parameters to stdout 47 ⟩
  ⟨ Initialize ion 'eta' parameter 48 ⟩
  L = sample_length/NUM_CELLS;      /* the universal length scale */
  MaxDepth = MaxDepth/L;      /* dimensionless */
  ⟨ Calculate crack and joint adjustment to permeability 49 ⟩
  critical_Cl_concentration = 0.000400 * Vsample * (1.0 − φ) * 2.5/35.4;
  if (anion[Cl].c[FIRST_CELL] > critical_Cl_concentration) {
      /* calculate T for [Cl_crit]/[Cl(x=0)] = erfc(X/2 sqrt(DT)) */
    concen_ratio = inv_erfc(critical_Cl_concentration/anion[Cl].c[0]);
    T_crit = SQR((rebar_depth/L)/(2.0 * concen_ratio));
    chloride_failure_year = 3.171 · 10^−08 * T_crit * SQR(L)/D_o;
    if (head.value > 0.0) {      /* correct diffusion estimate to include darcy penetration */
      darcy_velocity = (k_o/μ) * (ρ * g * head.value)/sample_length;
      A = 1.0/(4.0 * D_o * SQR(concen_ratio));
      perm_depth = rebar_depth + 1.0/(2.0 * A * darcy_velocity) − sqrt(rebar_depth/(A *
          darcy_velocity) + 1.0/(4.0 * SQR(A * darcy_velocity)));
      perm_time = perm_depth/darcy_velocity;
      chloride_failure_year = 3.171 · 10^−08 * perm_time;
    }
  }
  else chloride_failure_year = 1.0 · 10^+10;
```

```
  printf("\"Chloride␣failure␣(yr)\"\t\t␣%8.0lf\n", chloride_failure_year);
  perm_factor = MAX(k_o/3.0 · 10^-18, 1.0) * MAX(head.value/10., 1.0);
  if (crack_width.value ≡ 0.0) {
    crack_factor = 1.0;
  }
  else {
    if (sample_length − crack_depth > 0.05 * sample_length)
      crack_factor = sample_length/(sample_length − crack_depth);
    else crack_factor = 10.0;
  }
  ΔT = 1.0/(64.0 * perm_factor * crack_factor * anion[Cl].η * ϑ_0);
  sulfate_rate_ms = Youngs.value * SQR(β.value) * anion[SO4].c[FIRST_CELL] * CE.value *
      D_o/(roughness.value * γ.value * (1 − ν.value));      /* m/sec */
  sulfate_rate = sulfate_rate_ms * (L/D_o);      /* ΔX/ΔT */
  if (sulfate_rate > 0.0) sulfate_failure_year = 3.171 · 10^-08 * (MaxDepth * L)/sulfate_rate_ms;
  else sulfate_failure_year = 1.0 · 10^+10;
  printf("\"Sulfate␣failure␣(yr)\"\t\t␣%8.0lf\n", sulfate_failure_year);
     /* calculate number of days until termination */
  earliest_failure_day = 365. * MIN(sulfate_failure_year, chloride_failure_year);
     /* determine intervals between print outs */
  print_day_interval = MIN(earliest_failure_day, MaxDay.value)/PRINT_OUTS;
  print_day = print_day_interval;
  Pout = (k_o/(μ * D_o)) * ρ * g * head.value;
  dP = (k_o/(μ * D_o)) * ρ * g * (sample_length/NUM_CELLS);      /* change per cell */
  for (k = 0; k < NUM_SURFACES; k++) {
    ξ'[k] = ϑ_0;
    φ'^n[k] = φ;
    φ'^(n−1)[k] = φ;
    Ψ[k] = Pout * (real) (NUM_SURFACES − k − 1)/(NUM_SURFACES − 1);
  }
  ⟨ Initial estimate of pH 50 ⟩
}
```

42. The material parameters of the concrete must be determined based upon the information given by the user. Missing pieces of information must be calculated from established relationships. The *.is_default* portion of the material data determines whether the user specified the value.

```
⟨ Establish material parameters of concrete 42 ⟩ ≡
  if (D_∞.is_default ≠ TRUE) {      /* calculate porosity */
    if (wc.is_default ≡ TRUE) wc.value = (log10(10000. * D_∞.value) + 9.84)/6.0;
    if (k_∞.is_default ≡ TRUE)
        /* kinfty.value = 1.0E-18*(0.8904+.002525*exp(15.07*wc.value)); */
      k_∞.value = pow(10.0, (5.0 * wc.value − 21.0));
  }
  if (k_∞.is_default ≠ TRUE) {
    if (D_∞.is_default ≡ TRUE) {
      D_∞.value = 1.0 · 10^-04 * pow(10.0, 6.0 * wc.value − 9.84);
      if (wc.is_default ≡ TRUE) wc.value = (log10(10000. * D_∞.value) + 9.84)/6.0;
    }
  }
  if (wc.is_default ≠ TRUE) {      /* calculate diffusivity and porosity */
```

```
    if (D∞.is_default ≡ TRUE) D∞.value = 1.0 · 10^-04 * pow(10.0, 6.0 * wc.value − 9.84);
    if (k∞.is_default ≡ TRUE)
        /* kinfty.value = 1.0E-18*(0.8904+.002525*exp(15.07*wc.value)); */
      k∞.value = pow(10.0, (5.0 * wc.value − 21.0));
  }
  ϑ0 = D∞.value/anion[Cl].Df;
  if (ϑ0 < DLIMIT) ϑ0 = DLIMIT + 0.0001;
  if (ϑ0 < DLIMIT + 0.07 * SQR(.18)) φ = sqrt((ϑ0 − DLIMIT)/.07);
  else φ = .17326 + sqrt(.03002 − (.05832 + DLIMIT − ϑ0)/1.87);
  α = (3.2 * wc.value)/1.36 − φ * (1 + 3.2 * wc.value)/1.36;
  f = 1.0/(1 + 3.2 * wc.value);
  Vsample = 1000.0/φ;
  molesCH = α * (0.61) * f * Vsample/ρCH;
  F = 1.0/ϑ0;
  Do = D∞.value;
  ko = k∞.value;
  litre[0] = 0.001 * Vsample * φ;
  for (k = 1; k < NUM_SURFACES; k++) {
    sol_array[Ca][OH].s[k] = molesCH;
    litre[k] = 0.001 * Vsample * φ;
  }
  for (k = 0; k < NUM_CELLS; k++) {
    ξ[k] = ϑ0;
    φ^n[k] = φ;
  }
```

This code is used in section 41.

43. The routine *aci_211* estimates the strength, in psi, of the concrete in the individual computational elements. The estimate is based upon Table 5.3.4(a) of ACI 211.1-81 for air-entrained concrete. The strength can be adjusted to account for changes in porosity using:

$$psi'[i] = psi_o \frac{\ln(\phi^n[i])}{\ln(phi_o)}$$

```
⟨ Function declarations 23 ⟩ +≡
  void aci_211(void);
```

```
44. void aci_211()
  {
    int i;
    real ψ;
    ψ = pow(exp(wc.value − 3.575), −2.6817);
    for (i = 0; i < NUM_SURFACES; i++) strength[i] = ψ;
  }
```

45. The function *yyparse* sets the concentration in the concrete by putting the user-specified value into the . *c*[1] cell. The internal concentration is established by copying this value into the remaining cells. However, this procedure is not performed for OH and H.

```
⟨ Propagate ion concentration in concrete 45 ⟩ ≡
  for (i = 1; i < num_cations; i++) {
    cation[i].moles[0] = cation[i].c[0] * litre[0];
    cation[i].moles[1] = cation[i].c[1] * litre[1];
    for (k = 2; k < NUM_SURFACES; k++) {
      cation[i].c[k] = cation[i].c[1];
      cation[i].moles[k] = cation[i].c[k] * litre[k];
    }
  }
  for (j = 1; j < num_anions; j++) {
    anion[j].moles[0] = anion[j].c[0] * litre[0];
    anion[j].moles[1] = anion[j].c[1] * litre[1];
    for (k = 2; k < NUM_SURFACES; k++) {
      anion[j].c[k] = anion[j].c[1];
      anion[j].moles[k] = anion[j].c[k] * litre[k];
    }
  }
```

This code is used in section 41.

46. Report material parameters to *stdout*.

```
⟨ Preprocessor definitions 11 ⟩ +≡
#define GET_STAT(a) ((a.is_default ≡ TRUE) ? "DEFAULT" : "USER")
```

47.

```
⟨ Print parameters to stdout 47 ⟩ ≡
  printf("\"THICKNESS␣\"\t␣%8.5lf\t\"(m)␣␣␣␣␣␣\"\t\"%s\"\n", sample_length,
      GET_STAT(thickness));
  printf("\"DIFF␣␣␣␣␣␣\"\t␣%8.1le\t\"(m^2/sec)\"\t\"%s\"\n", D_o, GET_STAT(D_∞));
  printf("\"PERM␣␣␣␣␣␣\"\t␣%8.1le\t\"(m/sec)␣␣\"\t\"%s\"\n", 9.8 · 10^+06 * k_o,
      GET_STAT(k_∞));
  printf("\"WC␣␣␣␣␣␣␣␣\"\t␣%8.5lf\t\"␣␣␣␣␣␣␣␣␣\"\t\"%s\"\n", wc.value, GET_STAT(wc));
  printf("\"HEAD␣␣␣␣␣␣\"\t␣%8.5lf\t\"(m)␣␣␣␣␣\"\t\"%s\"\n", head.value, GET_STAT(head));
  printf("Sulfate␣Attack␣Parameters:\n");
  printf("\"YOUNGS␣␣␣␣\"\t␣%8.1le\t\"(N/m^2)␣␣\"\t\"%s\"\n", Youngs.value,
      GET_STAT(Youngs));
  printf("\"BETA␣␣␣␣␣␣\"\t␣%8.1le\t\"␣␣␣␣␣␣␣␣␣\"\t\"%s\"\n", β.value, GET_STAT(β));
  printf("\"CE␣␣␣␣␣␣␣␣\"\t␣%8.5lf\t\"(Mol/m^3)\"\t\"%s\"\n", CE.value, GET_STAT(CE));
  printf("\"ROUGHNESS␣\"\t␣%8.5lf\t\"␣␣␣␣␣␣␣␣␣\"\t\"%s\"\n", roughness.value,
      GET_STAT(roughness));
  printf("\"GAMMA␣␣␣␣␣\"\t␣%8.5lf\t\"(J/m^2)␣␣\"\t\"%s\"\n", γ.value, GET_STAT(γ));
  printf("\"POISSON␣␣␣\"\t␣%8.5lf\t\"␣␣␣␣␣␣␣␣␣\"\t\"%s\"\n", ν.value, GET_STAT(ν));
  printf("\n");
  if (MaxDepth < sample_length)
    printf("\"DEPTH␣␣␣␣␣\"\t␣%8.5lf\t\"(m)␣␣␣␣␣␣\"\t\"%s\"\n", MaxDepth, "USER");
  if (rebar_depth < sample_length)
    printf("\"REBAR␣␣␣␣␣\"\t␣%8.5lf\t\"(m)␣␣␣␣␣␣\"\t\"%s\"\n", rebar_depth, "USER");
  printf("\"TIME␣␣␣␣␣␣\"\t␣%8.0lf\t\"(day)␣␣␣␣\"\t\"%s\"\n", MaxDay.value,
      GET_STAT(MaxDay));
  if (crack_width.is_default ≡ TRUE ∧ neutral_axis_depth.is_default ≡ TRUE) {
    crack_depth = 0.25 * sample_length;
```

```
    crack_spacing = 2.0;
  }
  if (neutral_axis_depth.is_default ≡ FALSE)
    crack_depth = sample_length − neutral_axis_depth.value;
  if (crack_width.value > 0.0) printf("\"CRACK␣=␣%8.5lf␣AT␣%8.5lf␣DEPTH␣%8.5lf\"\n",
      crack_width.value, crack_spacing, crack_depth);
  if (joint_permeability > 0) {
    printf("\"JOINT␣PERM␣=␣%8.1le\"\n", joint_permeability);
    printf("\"JOINT=%8.5lf␣AT␣%8.5le␣UNTIL␣%8.5lf\"\n", joint_width, joint_spacing,
      joint_lifetime/365);
  }
```

This code is used in section 41.

48. Once the parameter D_o has been established, the value of η for each ion must be initialized.

```
⟨ Initialize ion 'eta' parameter 48 ⟩ ≡
  for (i = 0; i < num_cations; i++) cation[i].η = cation[i].D_f/D_o;
  for (j = 0; j < num_anions; j++) anion[j].η = anion[j].D_f/D_o;
```

This code is used in section 41.

49. The presence of cracks is reflected in κ which a multiplicative adjustment for permeability. In the absence of cracks $\kappa[i]$=1. The quantities *crack_width* and *crack_spacing* are specified by the user.

The equation for the permeability of cracks of width w spaced a distance a apart is

$$k = \frac{w^3}{12a}$$

```
⟨ Calculate crack and joint adjustment to permeability 49 ⟩ ≡
  crack_permeability = (SQR(crack_width.value)/12);
  x = 0;      /* x starts from inside surface */
  dx = sample_length/NUM_CELLS;
  for (k = NUM_CELLS − 1; k ≥ 0; k−−) {
    x += dx;
    if (x ≤ crack_depth) {
      κ[k] = k_o * (1. − (joint_width/joint_spacing) − (crack_width.value/crack_spacing));
        /* weighted length */
      κ[k] += joint_permeability * (joint_width/joint_spacing);
      κ[k] += crack_permeability * (crack_width.value/crack_spacing);
    }
    else {
      if (x − dx < crack_depth) {      /* crack ends in this cell */
        κ[k] = k_o * (1. − (joint_width/joint_spacing) − (crack_width.value/crack_spacing));
          /* weighted length */
        κ[k] += joint_permeability * (joint_width/joint_spacing);
        κ[k] = (x − crack_depth)/κ[k];
        κ[k] += (dx − (x − crack_depth))/(crack_permeability * (crack_width.value/crack_spacing));
        κ[k] /= dx;
        κ[k] = 1.0/κ[k];
      }
      else {
```

```
        κ[k] = k_o * (1. - (joint_width/joint_spacing));
        κ[k] += joint_permeability * (joint_width/joint_spacing);
      }
    }
    κ[k] = κ[k]/k_o;
  }
```

This code is used in section 41.

50. Need to make an initial estimate of the equilibrium concentrations and the pH of each cell.

```
⟨ Initial estimate of pH 50 ⟩ ≡
    /* make a rough guess for concentrations in cases of solids present */
  for (k = 0; k < NUM_SURFACES; k++) {
    for (i = 0; i < num_cations; i++) {
      for (j = 0; j < num_anions; j++) {
        if (sol_array[i][j].s[k] > 0.0) {
          m = sol_array[i][j].m;
          n = sol_array[i][j].n;
          if (m ≡ 0 ∨ n ≡ 0) {
            if (m ≡ 0) printf("m=0␣\n");
            if (n ≡ 0) printf("n=0␣\n");
            printf("\n\tProblem␣with␣%s%s\n", cation[i].name, anion[j].name);
            exit(0);
          }
          moles_cation = pow(sol_array[i][j].ksp,
              1./(m + n)) * pow(anion[j].valence/cation[i].valence, n/(n + m));
          moles_anion = moles_cation * (cation[i].valence/anion[j].valence);
          cation[i].c[k] = moles_cation;
          anion[j].c[k] = moles_anion;
          cation[i].moles[k] = cation[i].c[k] * litre[k];
          anion[j].moles[k] = anion[i].c[k] * litre[k];
        }
      }
    }
  }    /* calculated [OH] and [H] concentrations */
  for (k = 0; k < NUM_SURFACES; k++) {
    charge = 0.0;
    for (j = 1; j < num_anions; j++) charge += anion[j].valence * anion[j].c[k];
    for (i = 1; i < num_cations; i++) charge -= cation[i].valence * cation[i].c[k];
    anion[OH].c[k] = 0.5 * (-charge + sqrt(SQR(charge) + 4.0 * sol_array[H][OH].ksp));
    cation[H].c[k] = sol_array[H][OH].ksp/anion[OH].c[k];
    anion[OH].moles[k] = anion[OH].c[k] * litre[k];
    cation[H].moles[k] = cation[H].c[k] * litre[k];
  }
```

This code is used in section 41.

51. Inverse complementary error function. Find x given C such that $C = \mathrm{erfc}(x)$.

```
⟨ Function declarations 23 ⟩ +≡
  real inv_erfc(real);
```

52. Use lookup table from data in **Table 7.1** from Abramowitz and Stegun, Handbook of Mathematical Functions. Get bounds on y data and linear interpolate x data. The variable *frac* is the fraction of the gap between neighboring y values. **Note:** additions to the table require changes to the main routine to reflect the new *dimension* of the arrays.

```
real inv_erfc(real C)
{
  real x, frac, x_table[] = {0.00, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.90, 1.00, 1.20, 1.40,
      1.60, 1.80, 2.00}, y_table[] = {1.00, .888, .777, .671, .572, .480, .396, .322, .258, .203, .157,
      .090, .048, .024, .011, .005};
  int i;
  i = 16;
  while (i > 0 ∧ y_table[--i] < C) ;      /* find bounds */
  if (x < 15) {
    frac = (y_table[i] − C)/(y_table[i] − y_table[i + 1]);
    x = x_table[i] + frac * (x_table[i + 1] − x_table[i]);
  }
  else x = x_table[15];      /* if C > 2.0 simply use last x value */
  return x;
}
```

53. ADVECTION-DIFFUSION.

Ion Transport

At the core of 4SIGHT is the advection-diffusion equation to account for both diffusion of ions and Darcy flow of the pore solution due to hydrostatic head. The flux of ions due to both gradients in the ion concentration and to a volume average flow of pore solution is

$$\mathbf{j} = -D\nabla c + c\mathbf{u}$$

where $\mathbf{j}$ is the ion flux, D is the diffusivity, c is the ion concentration, and $\mathbf{u}$ is the volume-averaged velocity of the pore solution. The time dependent change in concentration is the negative divergence of the flux:

$$\frac{\partial c}{\partial t} = \nabla \cdot D\nabla c - \mathbf{u} \cdot \nabla c - c\nabla \cdot \mathbf{u}$$

Given a hydrostatic pressure head on a vertical column of porous media, the pore volume-averaged flow $\mathbf{v}_D$ is

$$\mathbf{v}_D = -\frac{k}{\mu}(\nabla p - \rho \mathbf{g})$$

For the hydrostatic heads considered here, the body force term, $\rho\mathbf{g}$ is non-negligible. This equation can be cast into the more familiar form, assuming a constant density pore fluid, using a modified pressure potential:

$$\psi = p - \rho g z$$

This gives the more familiar Darcy equation

$$\mathbf{v}_D = -\frac{k}{\mu}\nabla\psi$$

The volume-averaged velocity $\mathbf{u}$ can be related to the Darcy flow velocity:

$$\mathbf{v}_D = \phi\mathbf{u}$$

where ϕ is the porosity.

Finally, the above equations can be combined to give

$$\frac{\partial c}{\partial t} = \nabla \cdot D\nabla c + \frac{k}{\phi\mu}\nabla\psi \cdot \nabla c + c\nabla \cdot \frac{k}{\phi\mu}\nabla\psi$$

This equation gives the spatial and temporal behavior of the concentrations. To complete the calculations a means is needed to update the hydrostatic pressure potential, ψ.

Continuity Equation

The temporal behavior of ψ is calculated using the continuity equation:

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot \rho\mathbf{v}$$

where $\mathbf{v}$ is the intrinsic velocity of the pore solution. After averaging over the microstructure, the continuity equation becomes

$$\frac{\partial \phi}{\partial t} = -\nabla \cdot \mathbf{v}_D$$

This can be related back to pressure using the Darcy equation once again:

$$\frac{\partial \phi}{\partial t} = \nabla \cdot \frac{k}{\mu}\nabla\psi$$

54. Ion transport.

```
⟨ Function declarations 23 ⟩ +≡
  void ion_diffusion(void);
```

55. ion_diffusion.

```
void ion_diffusion( )
{
  int i, j, k;
  real cation_flux[10][NUM_SURFACES], anion_flux[10][NUM_SURFACES], upper_limit = 5.0 · 10^+04,
      lower_limit = 5.0 · 10^+03, max_flux_ratio;
  real tmp_cation[10][NUM_SURFACES], tmp_anion[10][NUM_SURFACES];
  ΔX[FIRST_CELL] = (real) (FIRST_CELL + 1) − sulfate_depth;
  max_flux_ratio = 0.0;
  for (i = 0; i < num_cations; i++) {
    for (k = FIRST_CELL + 1; k < NUM_SURFACES − 1; k++) {
      cation_flux[i][k] = cation[i].η * (ξ[k] * (cation[i].c[k + 1] − cation[i].c[k])/ΔX[k] − ξ[k − 1] *
          (cation[i].c[k] − cation[i].c[k − 1])/ΔX[k − 1])/(0.5 * (ΔX[k] + ΔX[k − 1]));
      cation_flux[i][k] −= (1./φ'^n[k]) * υ[k] * ((cation[i].c[k + 1] − cation[i].c[k − 1])/(ΔX[k] +
          ΔX[k − 1]));
      if (cation[i].c[k] > 0.0)
        max_flux_ratio = MAX(max_flux_ratio, ΔT * cation_flux[i][k]/cation[i].c[k]);
    }
  }
  for (j = 0; j < num_anions; j++) {
    for (k = FIRST_CELL + 1; k < NUM_SURFACES − 1; k++) {
      anion_flux[j][k] = anion[j].η * (ξ[k] * (anion[j].c[k + 1] − anion[j].c[k])/ΔX[k] − ξ[k − 1] *
          (anion[j].c[k] − anion[j].c[k − 1])/ΔX[k − 1])/(0.5 * (ΔX[k] + ΔX[k − 1]));
      anion_flux[j][k] −= (1./φ'^n[k]) * υ[k] * ((anion[j].c[k + 1] − anion[j].c[k − 1])/(ΔX[k] +
          ΔX[k − 1]));
      if (anion[j].c[k] > 0.0)
        max_flux_ratio = MAX(max_flux_ratio, ΔT * anion_flux[j][k]/anion[j].c[k]);
    }
  }
  if (max_flux_ratio > upper_limit) ΔT *= upper_limit/max_flux_ratio;
  if (max_flux_ratio < lower_limit) ΔT *= upper_limit/max_flux_ratio;
  for (k = FIRST_CELL + 1; k < NUM_SURFACES − 1; k++) {
    for (i = 0; i < num_cations; i++) cation[i].c[k] += ΔT * cation_flux[i][k];
    for (j = 0; j < num_anions; j++) anion[j].c[k] += ΔT * anion_flux[j][k];
  }
  k = NUM_SURFACES − 1;
  for (i = 0; i < num_cations; i++) cation[i].c[k] = cation[i].c[k − 1];
  for (j = 0; j < num_anions; j++) anion[j].c[k] = anion[j].c[k − 1];
  for (i = 0; i < num_cations; i++)
    for (k = FIRST_CELL; k < NUM_SURFACES; k++)
      cation[i].moles[k] = cation[i].c[k] * litre[k] * ΔX[k];
  for (j = 0; j < num_anions; j++)
    for (k = FIRST_CELL + 1; k < NUM_SURFACES; k++)
      anion[j].moles[k] = anion[j].c[k] * litre[k] * ΔX[k];
}
```

56. Chemical equilibrium.

Given the number density of ions in a computational element, determine if any of the ions should go in to/out of solution. Determine the pH and adjust H and OH, accordingly.

```
⟨ Function declarations 23 ⟩ +≡
  void chemical_equilibrium(boolean);
```

57. Minimizing function. This is the function to minimize for the *chemical_equilibrium* routine. The objective of *chemical_equilibrium* is to determine how many moles of salt should be leached/precipitated. Therefore, *minfunc* must adjust for the pore volume.

The equation *minfunc* is minimizing the square of

$$[C + \frac{mx}{V}]^m [A + \frac{nx}{V}]^n - k_{sp} = 0$$

where x is moles salt, m and n are stoichiometric ratios, and V is the pore solution volume.

```
  real minfunc(real, int, int, int);
  real minfunc(real x, int i, int j, int k)
  {
    real C, A;
    int m, n;
    m = sol_array[i][j].m;
    n = sol_array[i][j].n;
    C = cation[i].c[k];
    A = anion[j].c[k];
    return SQR(pow(C + m * x/(litre[k] * ΔX[k]), m) * pow(A + n * x/(litre[k] * ΔX[k]),
        n) − sol_array[i][j].ksp);
  }
```

58. *chemical_equilibrium*( ). Cycle through *sol_array* and determine if any ions should go in to/out of solution based upon the concentration, the solubility constant, and the presence of solid salt.

```
#define TOL 1.0 · 10^-06
  void chemical_equilibrium(boolean Change_Porosity)
  {
    int i, j, k, n, m, iterations;
    real w_max, δ;
    real xa, xb, xc, fa, fb, fc, moles, tmp;
    real dsolid;
    real Δlitre;      /* change in pore volume */
    real old_cOH;
    real charge, temp;
    boolean NEG_FLAG;
    for (k = FIRST_CELL + 1; k < NUM_SURFACES; k++) {
      iterations = 0;
      do {
        w_max = 0.0;
        for (j = 0; j < num_anions; j++) {
          for (i = 0; i < num_cations; i++) {
            if (¬(i ≡ 0 ∧ j ≡ 0)) {
```

```
        n = sol_array[i][j].n;
        m = sol_array[i][j].m;
        temp = pow(cation[i].c[k], (double) m) * pow(anion[j].c[k], (double) n);
        if ((sol_array[i][j].s[k] > 0.0) ∨ (temp > sol_array[i][j].ksp)) {
          xa = -MIN(cation[i].c[k], anion[j].c[k])/10.0;
          xb = -xa;
          xc = 0.0;
          mnbrak(&xa, &xb, &xc, &fa, &fb, &fc, minfunc, i, j, k);
          tmp = brent(xa, xb, xc, minfunc, TOL, &moles, i, j, k);
          if (fabs(moles) > 0.05)
            moles *= 0.10;      /* no drastic changes */
          if (sol_array[i][j].s[k] - moles < 0.0)
            moles = sol_array[i][j].s[k];      /* insufficient salt */
          if (cation[i].moles[k] + m * moles < 0.0)
            moles = -cation[i].moles[k]/m + 0.000001;
          if (anion[j].moles[k] + n * moles < 0.0)
            moles = -anion[j].moles[k]/n + 0.000001;
          Δlitre = 0.001 * moles * sol_array[i][j].molar_density/ΔX[k];
          if (litre[k] + Δlitre < 0.0) {
            Δlitre = -litre[k];      /* insufficient pore space */
            moles = Δlitre * ΔX[k] * 1000.0/sol_array[i][j].molar_density;
          }
          cation[i].moles[k] += m * moles;
          anion[j].moles[k] += n * moles;
          if (Change_Porosity ≡ TRUE) {
            litre[k] += Δlitre;
            sol_array[i][j].s[k] -= moles;
          }
          δ = fabs(moles);
          w_max = MAX(δ, w_max);
        }
      }
    }
  }      /* readjust the concentrations */
  φ'^n[k] = 1000.0 * litre[k] * ΔX[k]/Vsample;
  litre[k] = MAX(litre[k], 0.0001);      /* avoid zero volume problems */
  for (i = 0; i < num_cations; i++)
    cation[i].c[k] = cation[i].moles[k]/(litre[k] * ΔX[k]);
  for (j = 0; j < num_anions; j++)
    anion[j].c[k] = anion[j].moles[k]/(litre[k] * ΔX[k]);
      /* Determine [OH] and [H]concentration */      /* store old value */
  old_cOH = anion[OH].c[k];
  charge = 0.0;
  for (j = 1; j < num_anions; j++) charge += anion[j].valence * anion[j].c[k];
  for (i = 1; i < num_cations; i++) charge -= cation[i].valence * cation[i].c[k];
  anion[OH].c[k] = 0.5 * (-charge + sqrt(SQR(charge) + 4.0 * sol_array[H][OH].ksp));
  anion[OH].moles[k] = anion[OH].c[k] * litre[k] * ΔX[k];
  if (fabs((anion[OH].c[k] - old_cOH)/old_cOH) > 1.00) {
    w_max = 1.0;
    anion[OH].c[k] = 0.5 * (anion[OH].c[k] + old_cOH);
    anion[OH].moles[k] = anion[OH].c[k] * litre[k] * ΔX[k];
```

```
        }
        cation[H].c[k] = sol_array[H][OH].ksp/anion[OH].c[k];
        cation[H].moles[k] = cation[H].c[k] * litre[k] * ΔX[k];
      } while ((w_max > 0.0001 ∨ anion[OH].c[k] < 0.0) ∧ iterations++ < 50);
    }     /* extrapolate φ'^n[FIRST_CELL] value */
    φ'^n[FIRST_CELL] = φ'^n[FIRST_CELL + 1] − ΔX[FIRST_CELL] * (φ'^n[FIRST_CELL + 2] −
        φ'^n[FIRST_CELL + 1]);
    φ'^n[FIRST_CELL] = MAX(φ'^n[FIRST_CELL], 0.0);     /* extrapolate φ'^n[NUM_CELL] value */
    φ'^n[NUM_CELLS] = 2.0 * φ'^n[NUM_CELLS − 1] − φ'^n[NUM_CELLS − 2];
      /* extrapolate litre[FIRST_CELL] value */
    litre[FIRST_CELL] = 0.001 * Vsample * φ'^n[FIRST_CELL];
  }
```

59. Minimization. The routine *mnbrak* brackets a minimum.

```
⟨ Function declarations 23 ⟩ +≡
  void mnbrak(real *, real *, real *, real *, real *, real *, real(*func)(real, int, int, int), int,
      int, int);
#include "mnbrak.c"
```

60. Parabolic approximation. The routine *brent* uses parabolic approximation to determine the minimum given three points bracketing the minimum.

```
⟨ Function declarations 23 ⟩ +≡
  real brent(real, real, real, real(*func)(real, int, int, int), real, real *, int, int, int);
#include "brent.c"
```

61. Advance sulfate front.

```
⟨ Function declarations 23 ⟩ +≡
  void advance_sulfate_front( );
```

```
62. void advance_sulfate_front( )
  {
    int i, j, k, new_first_cell;
    sulfate_depth += sulfate_rate * ΔT;
    if (sulfate_depth > (real) (FIRST_CELL + 1) ∨ ΔX[FIRST_CELL] < 0.10) {
      new_first_cell = (int) (sulfate_depth + 0.4);      /* penetration beyond next cell */
        /* copy info from old first cell into new first cell */
      for (k = FIRST_CELL + 1; k ≤ new_first_cell; k++) {
        for (i = 0; i < num_cations; i++) cation[i].c[k] = cation[i].c[FIRST_CELL];
        for (j = 0; j < num_anions; j++) anion[j].c[k] = anion[j].c[FIRST_CELL];
        for (i = 0; i < num_cations; i++)
          for (j = 0; j < num_anions; j++) sol_array[i][j].s[k] = sol_array[i][j].s[FIRST_CELL];
      }
      Ψ[new_first_cell] = Ψ[FIRST_CELL];
      FIRST_CELL = new_first_cell;
      ΔX[FIRST_CELL] = (real) (FIRST_CELL + 1) − sulfate_depth;
    }
  }
```

63. Chloride penetration.

The ratio of the mass of chloride ions to the mass of concrete equal to 0.0004 is the depth of chloride penetration. The ration equals $10^{-3}[Cl^-]A_{Cl}\phi/((1-\phi)2.5)$. $A_{Cl}(gm/mole)$ is the gram atomic mass of chlorine, ϕ is porosity, and 2.5 is the density of concrete.

```
⟨ Global variables 4 ⟩ +≡
  real Cl_ratio[NUM_SURFACES];
```

64. ⟨ Assess chloride penetration 64 ⟩ ≡

```
for (k = FIRST_CELL; k < NUM_SURFACES; k++)
  Cl_ratio[k] = 35.5 * anion[Cl].moles[k]/(2.5 * (1.0 - φ'^n[k]) * Vsample);
k = NUM_SURFACES - 1;
while (k ≥ 0 ∧ Cl_ratio[k] < 0.0004) k--;
if (k < 0) chloride_depth = 0.0;      /* k at lower limit */
else {
  if (k ≡ FIRST_CELL) {      /* account for distance equals ΔX */
    chloride_depth = ΔX[FIRST_CELL] * (Cl_ratio[k] - 0.0004)/(Cl_ratio[k] - Cl_ratio[k + 1]);
    chloride_depth += (real) FIRST_CELL;
  }
  else chloride_depth = (real) k + (Cl_ratio[k] - 0.0004)/(Cl_ratio[k] - Cl_ratio[k + 1]);
}
```

This code is used in section 10.

65. Update Pressures.

The pressures are updated using the continuity equation:

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot \rho \mathbf{v}$$

where $\mathbf{v}$ is the intrinisic pore fluid velocity and ρ is the pore fluid density. An equation for the bulk material can be obtained from a volume average over a represetative volume V:

$$\frac{1}{V}\int_V \frac{\partial \rho}{\partial t}\, d^3\mathbf{x} = -\frac{1}{V}\int_V \nabla \cdot \rho \mathbf{v}\, d^3\mathbf{x}$$

For these equations, assume ρ is a constant since water is virtually incompressible. Rearranging and simplifying the above equation gives:

$$\frac{\partial}{\partial t}\frac{1}{V}\int_V \rho\, d^3\mathbf{x} = -\nabla \cdot \frac{1}{V}\int_V \rho \mathbf{v}\, d^3\mathbf{x}$$

Since ρ is zero outside the pore volume, if V_p represents only the pore volume then the above equation simplifies to

$$\frac{\partial \phi}{\partial t} = -\nabla \cdot \frac{\phi}{V_p}\int_{V_p} \mathbf{v}\, d^3\mathbf{x}$$

which finally gives

$$\frac{\partial \phi}{\partial t} = -\nabla \cdot \phi \mathbf{u} = -\nabla \cdot \mathbf{v}_D$$

where $\mathbf{v}_D$ is the Darcy velocity. Substituting for the Darcy velocity gives

$$\frac{\partial \phi}{\partial t} = \nabla \cdot \frac{k}{\mu}\nabla p$$

⟨ Function declarations 23 ⟩ +≡

```
void update_pressures(void);
```

66. update_pressures.

```
void update_pressures( )
{
  int k, iteration;
  real perm[NUM_CELLS], tmp[NUM_SURFACES], tol = 0.0005, ϵ;
  for (k = FIRST_CELL; k < NUM_CELLS; k++) perm[k] = κ[k] * CUB(ξ[k]/ϑ0);
  Ψ[FIRST_CELL] = Pout + (NUM_CELLS − 1 + ΔX[FIRST_CELL]) * dP;
  Ψ[NUM_SURFACES − 1] = 0;
  iteration = 0;
  do {
    for (k = FIRST_CELL + 1; k < NUM_SURFACES − 1; k++) {
      tmp[k] = perm[k] * Ψ[k + 1]/ΔX[k] + perm[k − 1] * Ψ[k − 1]/ΔX[k − 1];
        /* **** tmp[k] -= 0.5*(DX[k-1]+DX[k])*(PHIn[k]-PHIn[k-1])/DT; *** */
      tmp[k] *= 1.0/(perm[k]/ΔX[k] + perm[k − 1]/ΔX[k − 1]);
    }
    ϵ = 0.0;
    for (k = FIRST_CELL + 1; k < NUM_SURFACES − 1; k++) {
      if (Ψ[k] > 0.0) ϵ = MAX(ϵ, fabs(Ψ[k] − tmp[k])/Ψ[k]);
```

```
        Ψ[k] = tmp[k];
      }
    } while (ε > tol ∧ iteration++ < 1000);
    for (k = 0; k < NUM_CELLS; k++)
      υ[k] = −κ[k] * CUB(ξ[k]/ϑ0) * (Ψ[k + 1] − Ψ[k])/ΔX[k];
  }
```

67. Copy the new porosities into the $n-1$ values.

```
⟨ Update the porosities 67 ⟩ ≡
  for (k = FIRST_CELL; k < NUM_SURFACES; k++) φ'^{n−1}[k] = φ'^n[k];
```

This code is used in section 9.

68. Adjust physical parameters.

Due to dissolution/precipitation.

```
⟨ Adjust physical parameters 68 ⟩ ≡
  for (k = FIRST_CELL; k < NUM_CELLS; k++) {
    φ^n[k] = 0.5 * (φ'^n[k] + φ'^n[k + 1]);      /* interpolate phi' values */
    ϑ = DLIMIT + 0.07 * SQR(φ^n[k]);
    if (φ^n[k] > 0.180) ϑ += 1.8 * SQR(φ^n[k] − .180);
    if (φ^n[k] > φ)      /* leaching */
      ξ[k] = ϑ_0 + 5.0 * (ϑ − ϑ_0);
    else      /* precipitation */
      ξ[k] = ϑ;
  }
  ξ'[FIRST_CELL] = ξ[FIRST_CELL] − (ΔX[FIRST_CELL]/(1.0 + ΔX[FIRST_CELL])) * (ξ[FIRST_CELL +
      1] − ξ[FIRST_CELL]);
  for (k = FIRST_CELL + 1; k < NUM_SURFACES − 1; k++) ξ'[k] = 0.5 * (ξ[k] + ξ[k − 1]);
  ξ'[NUM_SURFACES − 1] = 1.5 * ξ[NUM_CELLS − 1] − 0.5 * ξ[NUM_CELLS − 2];
```

This code is used in section 9.

69. Proper time.

The clock time must be advanced

```
⟨ Global variables 4 ⟩ +≡
  real Time = 0;      /* dimensionless cumulative time */
  real Day = 0;       /* cumulative time (day) */
```

70.
```
⟨ Advance global clock 70 ⟩ ≡
  Time += ΔT;
  Day = Time * SQR(L)/(D_o * 86400);
```
This code is used in section 9.

71. Check time dependent variables.

```
⟨ Check time dependencies 71 ⟩ ≡
  if (Day > joint_lifetime ∧ joint_is_specified ≡ TRUE) {
    joint_change_flag = TRUE;
  }
```
This code is used in section 10.

72.
```
⟨ Assess simulation termination 72 ⟩ ≡
  if (Day > MaxDay.value) termination = TRUE;
  if (sulfate_depth > MaxDepth) termination = TRUE;
  if ((chloride_depth * L) > rebar_depth) termination = TRUE;
```
This code is used in section 10.

73. OUTPUT.

The following are output routines.

74. Print the state of the system for the first 4 concentraction surfaces.

```
⟨ Print initial system state 74 ⟩ ≡
  printf("\n\nInitial␣state␣of␣system:\n\n");
  printf("\"␣ION␣\"\t\"EXTERNAL\"\t\"INTERNAL\"\n");
  for (i = 0; i < num_cations; i++) {
    if (cation[i].c[FIRST_CELL] ≠ 0 ∨ cation[i].c[FIRST_CELL + 1] ≠ 0) {
      printf("\"%4s:\"\t", cation[i].name);
      for (k = FIRST_CELL; k < FIRST_CELL + 2; k++) printf("%8.5lf\t", cation[i].c[k]);
      printf("\n");
    }
  }
  for (j = 0; j < num_anions; j++) {
    if (anion[j].c[FIRST_CELL] ≠ 0 ∨ anion[j].c[FIRST_CELL + 1] ≠ 0) {
      printf("\"%4s:\"\t", anion[j].name);
      for (k = FIRST_CELL; k < FIRST_CELL + 2; k++) printf("%8.5lf\t", anion[j].c[k]);
      printf("\n");
    }
  }
  printf("\"␣␣pH:\"\t");
  for (k = FIRST_CELL; k < FIRST_CELL + 2; k++) printf("%8.5lf\t", -log10(cation[H].c[k]));
  printf("\n");
  printf("\n");
```

This code is used in section 7.

75. Print header for permeability, sulfate penetration, and chloride penetration depths.

```
⟨ Print header 75 ⟩ ≡
  printf("\"Day\"\t\"L\"\t\"K␣␣\"\t\"D␣␣␣␣\"\t\"SO4\"\t\"Cl\"\t\"Flux\"\t␣␣\"pH\"\n");
  printf("\"␣␣␣\"\t\"m\"\t\"m/s\"\t\"m^2/s\"\t\"m␣␣\"\t\"m␣\"\t\"ml/dy/m2\"\n");
  printf("\n");
```

This code is used in section 2.

76. Print intermediate results for permeability, sulfate and chloride penetration depths. The bulk values of permeability and diffusivity must be calculated here.

```
⟨ Print intermediate results 76 ⟩ ≡
  if (Day ≡ 0.0 ∨ Day > print_day) {
    print_day += print_day_interval;
    printf("%ld␣\t%6.3lf", (long) Day, (ΔX[FIRST_CELL] + NUM_CELLS − FIRST_CELL − 1) * L);
    k_B = ΔX[FIRST_CELL] * CUB(ϑ_0/ξ[FIRST_CELL])/κ[FIRST_CELL];
    D_B = ΔX[FIRST_CELL] * (ϑ_0/ξ[FIRST_CELL]);
    for (k = FIRST_CELL + 1; k < NUM_CELLS; k++) {
      k_B += CUB(ϑ_0/ξ[k])/κ[k];
      D_B += ϑ_0/ξ[k];
    }
    k_B = (ΔX[FIRST_CELL] + NUM_CELLS − FIRST_CELL − 1) * k_o/k_B;
```

```
    D_B = (ΔX[FIRST_CELL] + NUM_CELLS − FIRST_CELL − 1) * D_o/D_B;
    printf("\t%6.1le", 9.8 · 10^+06 * k_B);
    printf("\t%6.1le", D_B);
    printf("\t%6.3lf\t%6.3lf", sulfate_depth * L, chloride_depth * L);
    printf("\t%6.3lf", 1000. * 86.4 * v[NUM_CELLS − 1] * D_o * Vsample/SQR(L));
    printf("\t%6.1lf\n", −log10(cation[H].c[NUM_SURFACES − 1]));
  }
```

This code is used in section 2.

77. Print termination information to include the day of termination and the depth of chloride and sulfate penetration.

```
⟨ Print termination information 77 ⟩ ≡
  printf("\n\n");
  if (Day > MaxDay.value ∧ joint_change_flag ≡ FALSE) printf("Exceeded␣TIME␣limit.\n\n");
  if (Day > MaxDay.value ∧ joint_change_flag ≡ TRUE)
    printf("Exceeded␣JOINT␣LIFETIME.\n\n");
  if (sulfate_depth > MaxDepth) printf("Sulfate␣Failure.\n\n");
  if (chloride_depth * L > rebar_depth) printf("Chloride␣Failure.\n\n");
  printf("\"T\"\t%8.3lf\n", Time);
  printf("\"Day\"\t%8.0lf\n", Day);
  printf("\"SO4␣(m)\"\t%8.3lf\n", sulfate_depth * L);
  printf("\"Cl␣(m)\"\t%8.3lf\n", chloride_depth * L);
  printf("\n\n");
```

This code is used in section 2.

78. Print final system state for any desired ion or salt.

```
⟨ Print desired output 78 ⟩ ≡
  printf("Final␣System␣state:\n\n");
  printf("\"L(m)\"␣\t");
  printf("\"Psi\"␣\t");
  printf("\"vD\"␣\t");
  printf("\"xi\"␣\t");
  printf("\"phi\"\t");
  printf("\"pH\"␣\t");
  printf("\"fc\"␣\t");
  for (i = 0; i < num_cations; i++)
    if (cation[i].output_flag ≡ TRUE) printf("\"%s\"␣\t", cation[i].name);
  for (j = 0; j < num_anions; j++)
    if (anion[j].output_flag ≡ TRUE) printf("\"%s\"␣\t", anion[j].name);
  for (i = 0; i < num_cations; i++)
    for (j = 0; j < num_anions; j++)
      if (sol_array[i][j].output_flag ≡ TRUE)
        printf("\"%s%s\"␣\t", cation[i].name, anion[j].name);
  printf("\n");
  for (k = 0; k < NUM_SURFACES; k++) {
    printf("%7.4lf\t", L * k);
    printf("%7.3lf\t", Ψ[k]);
    if (k < NUM_CELLS) {
      printf("%6.3lf␣\t", v[k]);
```

```
    }
    else {
      printf("%6.3lf␣\t", v[NUM_CELLS − 1]);
    }
    printf("%6.4lf␣\t", ξ′[k]);
    printf("%6.4lf␣\t", φ′ⁿ[k]);
    printf("%6.3lf␣\t", 14.0 + log10(anion[0].c[k]));
    printf("%6.0lf␣\t", strength[k] ∗ (log(φ′ⁿ[k])/log(φ)));
    for (i = 0; i < num_cations; i++)
      if (cation[i].output_flag ≡ TRUE) printf("%6.4lf␣\t", cation[i].c[k]);
    for (j = 0; j < num_anions; j++)
      if (anion[j].output_flag ≡ TRUE) printf("%6.4lf␣\t", anion[j].c[k]);
    for (i = 0; i < num_cations; i++)
      for (j = 0; j < num_anions; j++)
        if (sol_array[i][j].output_flag ≡ TRUE) printf("%6.3lf␣\t", sol_array[i][j].s[k]);
    printf("\n");
  }
  elapsed_time = biostime(GET_TIME, 0 L) − elapsed_time;
  printf("\"Elapsed␣time␣(sec):\"\t%6.1f\n", elapsed_time/CLK_TCK);
```

This code is used in sections 2, 24, and 26.