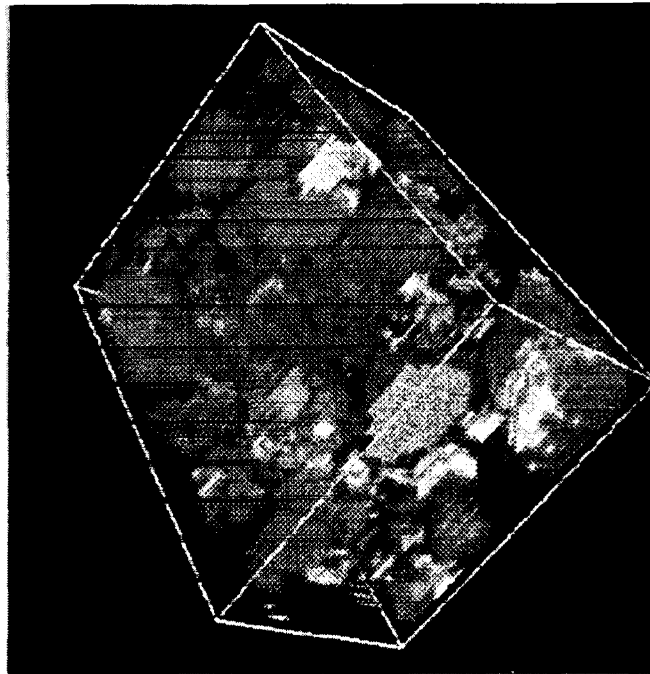
NISTIR 5977

Guide to Using CEMHYD3D: A Three-Dimensional Cement Hydration and Microstructure Development Modelling Package

Dale P. Bentz



Building and Fire Research Laboratory
Gaithersburg, Maryland 20899

NIST

United States Department of Commerce
Technology Administration
National Institute of Standards and Technology

NISTIR 5977

Guide to Using CEMHYD3D: A Three-Dimensional Cement Hydration and Microstructure Development Modelling Package

Dale P. Bentz

February 1997
Building and Fire Research Laboratory
National Institute of Standards and Technology
Gaithersburg, Maryland 20899

U.S. Department of Commerce
William M. Daley, *Secretary*
Technology Administration
Mary L. Good, *Under Secretary for Technology*
National Institute of Standards and Technology
Arati Prabhakar, *Director*

ABSTRACT

This user's manual provides documentation and computer program listings for the three-dimensional cement hydration and microstructure development model developed at the National Institute of Standards and Technology. The following topics are covered in this documentation: acquisition and processing of two-dimensional scanning electron microscope and x-ray images; creation of a starting three-dimensional microstructure based on a measured particle size distribution for the cement powder and information extracted from the two-dimensional image; and execution of the cement hydration and microstructure development program. Complete program listings and example datafiles are provided in the appendices of this documentation and the software is also available for downloading via anonymous ftp.

Keywords: Building technology, cement hydration, computer modelling, heat of hydration, image processing, microstructure, percolation, simulation.

BLANK PAGE

Contents

Abstract iii

List of Figures vii

List of Tables viii

1 Introduction 1

2 Two-dimensional Imaging of Cement Particles 3
- 2.1 Scanning electron microscopy image acquisition 3
- 2.2 Image segmentation and median filtering 3
- 2.3 Measurement of autocorrelation functions 4

3 Two-dimensional to Three-dimensional Conversion 7
- 3.1 Generation of spherical particles following measured PSD 7
- 3.2 Filtering of random noise particle image 9
- 3.3 Correction of hydraulic radius 9

4 Three-dimensional Cement Hydration Model 11
- 4.1 Inputs 11
- 4.2 Model execution 13
- 4.3 Outputs 13

5 Example Applications 14
- 5.1 Set Point- Percolation of Solids 14
- 5.2 Percolation Threshold of Capillary Pore Space 14
- 5.3 Prediction of Adiabatic Heat Signature 15

6 Acknowledgements 17

7 References 18

A Computer programs for two-dimensional imaging 20
- A.1 Listing for statsimp.c 20
- A.2 Listing for corrcalc.c 22
- A.3 Listing for corrxy2r.c 24

B Computer programs for two-dimensional to three-dimensional conversion 27
- B.1 Listing for genpart3d.c 27
- B.2 Listing for rand3d.f 46
- B.3 Listing for stat3d.c 52
- B.4 Listing for sinter3d.c 55

C Computer programs for three-dimensional cement hydration model 71
- C.1 Code for assessing percolation of pore space 71
- C.2 Code for assessing percolation of total solids- set point 74
- C.3 Code for random number generation 79

C.4 Listing for hydreal3d.c 80
C.5 Listing for disreal3d.c 108

D Example input datafiles for program execution **138**
D.1 Example input datafile for use with hydration model 138
D.2 Example input datafile for assessing percolation of solids 138
D.3 Example input datafile for assessing percolation of pore space 139

List of Figures

1 Flow diagram summarizing experimental and modelling program for predicting cement performance. 2

2 Flowchart specifying segmentation algorithm for selecting pixel phase values. 5

3 Final segmented two-dimensional image of Cement 116 issued by the Cement and Concrete Reference Laboratory (NIST). Phases from brightest to darkest are: C_3A, gypsum, C_4AF, C_3S, C_2S, and porosity. Image is approximately 250 μm $\times$ 200 μm. 6

4 Flowchart specifying filtering algorithm for assigning pixel phase values to the three-dimensional starting cement microstructure image. 10

5 Connected total solids volume fraction vs. degree of hydration for a cement paste with a w/c ratio of 0.4. 15

6 Connected capillary porosity fraction vs. volume fraction of porosity for a cement paste with a w/c ratio of 0.4. 15

7 Experimentally measured and model predicted (solid line) adiabatic temperature rise for an ordinary portland cement concrete containing 72% aggregates on a mass basis. 17

List of Tables

1 Steps in Execution of Three-Dimensional Cement Microstructure Model . . . 1
2 Volume in Pixels Occupied by Spheres of Various Diameters 8

1 Introduction

A package of computer programs for simulating cement hydration and cement paste microstructure development in **three dimensions** has been developed. A three-dimensional representation of microstructure is necessary for the computation of percolation and physical properties for comparison to experiment. While other recent publications have focused on the validation of the computer model [1] and various extensions [2], the purpose of this report is to provide a detailed documentation of the computer codes so that other researchers may employ them in studying problems specific to their own interests. It is strongly suggested that potential users also obtain copies of references [1] and [2] to obtain a better understanding of the technical basis underlying these documented computer programs. The experimental program which has been pursued for validating the computer models is summarized in Fig. 1. The steps required for going from a two-dimensional image to a simulation of three-dimensional microstructure development, along with the corresponding computer program names, are provided in Table I. These programs will be described in detail in sections 2 through 4 of this user's manual.

The computer codes are available via anonymous ftp as part of the Computer Integrated Knowledge System for High-Performance Concrete [3] being developed in the Building and Fire Research Laboratory at the National Institute of Standards and Technology. They may be accessed in the /ftp/pub/CEMHYD3D subdirectory from edsel.cbt.nist.gov (129.6.104.138), by logging in as user "anonymous" and providing your e-mail address as the password. Postscript versions of this manual and reference [1] are also available in the /ftp/pub/CEMHYD3D/manual subdirectory.

Table 1: Steps in Execution of Three-Dimensional Cement Microstructure Model

Step	Programs
Acquire and process two-dimensional image	statsimp.c
Determine autocorrelation for $C_3S + C_2S$ C_3S C_3A or C_4AF	corrcalc.c corrxy2r.c
Generate 3-D particle image	genpart3d.c
Distribute phases in 3-D image for silicates/aluminates C_3S/C_2S C_3A/C_4AF	rand3d.f stat3d.c sinter3d.c
Execute hydration model	disreal3d.c

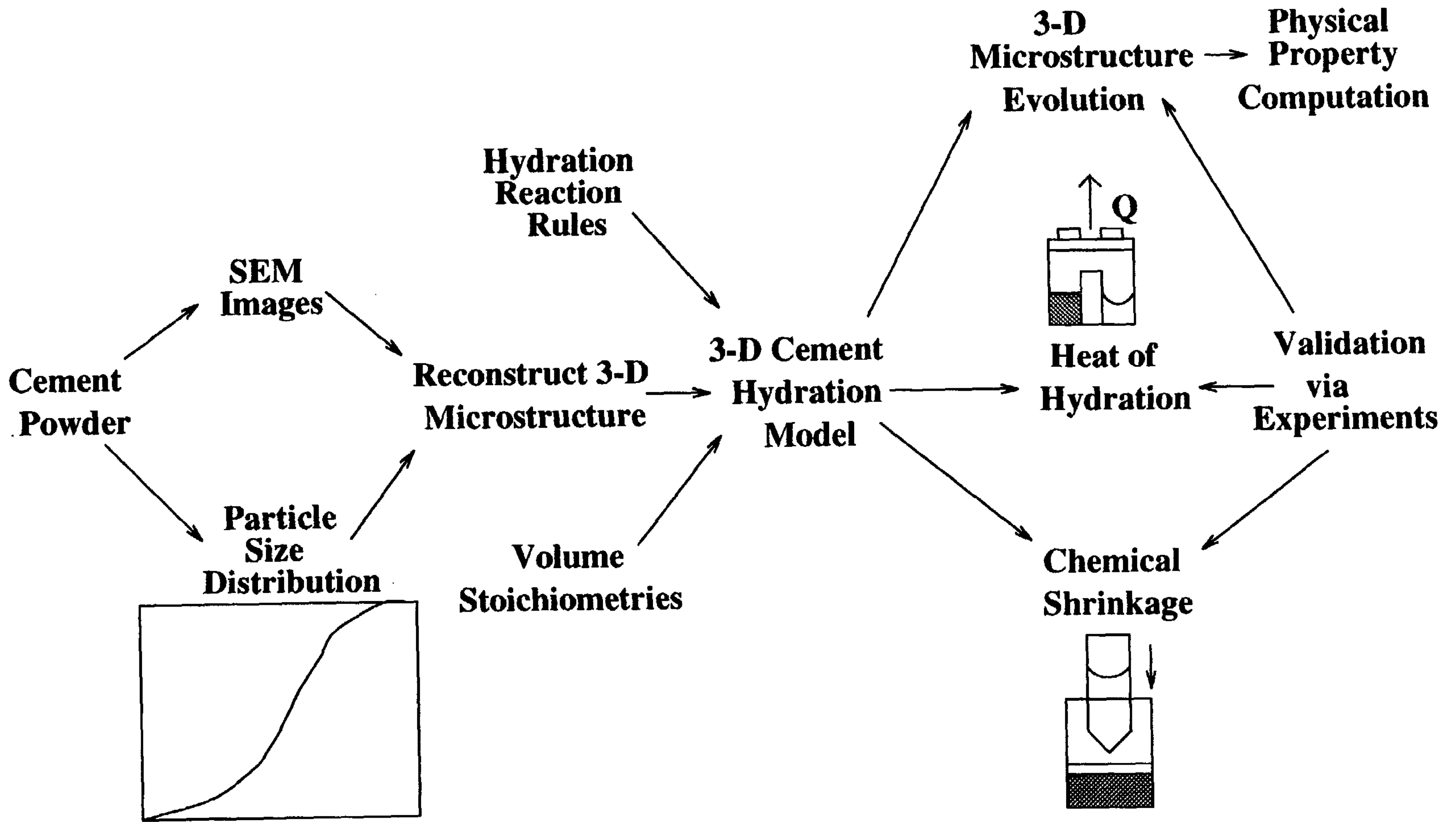


Figure 1: Flow diagram summarizing experimental and modelling program for predicting cement performance.

2 Two-dimensional Imaging of Cement Particles

In order to accurately model the hydration characteristics of a portland cement, it is necessary to assure that the original three-dimensional representation of the cement particles in water is as realistic as possible. With this in mind, experimental and computational techniques have been developed to characterize the multi-phase spatial structure of the starting cement powder and create a three-dimensional representation which reliably represents the following characteristics: particle size distribution, phase volume fractions, phase surface area fractions, and phase autocorrelation structure. Determinations of the latter three of these characteristics are based on image analysis of two-dimensional images of the starting cement powder, as described in this section.

2.1 Scanning electron microscopy image acquisition

Because different laboratories will have different scanning electron microscopy (SEM) equipment, only generalities can be presented in this portion of the documentation. Beginning with material preparation, the cement powder of interest is sampled and approximately 25 grams are mixed with a low viscosity epoxy resin [4] to form an almost dry paste. The epoxy is subsequently cured at 60 °C for 24 hours and the specimen is then cut, polished, and coated with a thin film of carbon in preparation for SEM viewing. Careful sample preparation is critical to the successful imaging of the fine cement particles [5, 6].

The sample is then viewed in the SEM and analyzed based on a combination of the backscattered electrons and X-rays emitted due to the specimen-primary electron beam interactions. For viewing cement powders, typical settings are an accelerating voltage of 12 kV and probe currents of 2 nA and 10 nA for the backscattered electron and X-ray imaging, respectively [6]. Typically, in addition to the backscattered electron image, X-ray images are collected for Ca, Si, S, Al, and Fe. These images are then processed and combined to determine the distribution of phases amongst the cement particles comprising the image.

The major phases present in a typical portland cement are: tricalcium silicate (C_3S), dicalcium silicate (C_2S), tricalcium aluminate (C_3A), tetracalcium aluminoferrite (C_4AF), and gypsum ($C\bar{S}H_2$) [1]. Because brightness in the backscattered electron image is proportional to the average atomic number of a phase, the C_4AF phase shows up as the brightest phase, followed by C_3S, C_3A, C_2S, $C\bar{S}H_2$, and resin-filled voids. Unfortunately, the intensities for the C_3A and C_2S are similar enough that a complete and accurate separation of phases based simply on the backscattered electron image is not possible. Fortunately, the X-ray images can provide valuable supplemental information concerning the phases comprising the individual cement particles in the image.

2.2 Image segmentation and median filtering

The key to combining information contained in the X-ray and backscattered electron images is that they all have been obtained in viewing the same area of the specimen. Thus, at any pixel (location) in the image, one has six signals available, the backscattered electron

[1]Conventional cement chemistry notation is used throughout this documentation: C=CaO, S=SiO_2, A=Al_2O_3, F=Fe_2O_3, $\bar{S}$=SO_3, and H=H_2O.

intensity and the five acquired X-ray signals. These signals can be interpreted as a set to determine the underlying phase present at each pixel comprising the image. The general algorithm for this is illustrated in Fig. 2. Here, the ratio of calcium to silicon (Ca/Si) is employed to separate the C_3S from the C_2S. Alternatively, the intensity of the backscattered electron image could be employed for this purpose, as discussed in the previous section. In addition, the alkali sulfates are identified as gypsum using the algorithm in Fig. 2. However, if X-ray images are also collected for K and Na, the alkali sulfates can be readily separated from the gypsum by comparing the K and Na intensities to user specified threshold values. The threshold values, X^*, to use for each X-ray image can generally be determined by viewing the image's greylevel (intensity) histogram [7]. This histogram is simply a plot of the number of pixels in an image corresponding to each possible greylevel value (with greylevels typically ranging from 0 to 255). The intensities corresponding to pixels where the element of interest is present will generally comprise either a separate peak or a peak shoulder of this histogram.

Once this initial segmentation is performed, the resulting image still contains a considerable amount of noise which must be removed. To do this, a type of median filtering algorithm is employed. Here, a small square, typically 5 pixels by 5 pixels, is centered at each pixel in the processed image. For all non-void pixels, the number (proportion) of pixels of each phase in this square is tabulated and the current pixel value replaced by the majority phase present in the square, if this majority proportion exceeds 0.5, to create a new "final" processed image. Figure 3 shows such a final processed image for a type I ordinary portland cement [2]. Using simple point counting procedures (program **statsimp.c** given in Appendix A), the area percentage of a phase(s) and the percentage of a phase(s) comprising the particle surfaces (perimeter percentage) can be readily determined for these images.

2.3 Measurement of autocorrelation functions

In addition to the area and perimeter percentages, for the three-dimensional reconstruction algorithms to be described below, it is also necessary to measure the autocorrelation functions for the individual phases and certain combinations of phases. For an $M \times N$ image, the autocorrelation function, $S(x,y)$, takes the form:

$$S(x,y) = \sum_{i=1}^{M-x} \sum_{j=1}^{N-y} \frac{I(i,j) \times I(i+x, j+y)}{(M-x) \times (N-y)} \tag{1}$$

where I(x,y) has a value of 1 if the pixel located at (x,y) meets the user criteria (e.g., contains the phase of interest) and 0 otherwise. The program **corrcalc.c**, contained in Appendix A, implements this procedure for a 2-D image with the following phase assignments:

porosity, gypsum - 0

C_3S - 1

C_2S - 2

C_3A - 3

C_4AF - 4.

[2]A complete set of images illustrating this procedure can be viewed in section 2 of a chapter on cement hydration modelling for an online monograph available at:
http://ciks.cbt.nist.gov/~ bentz/chapter/hydmod.html.

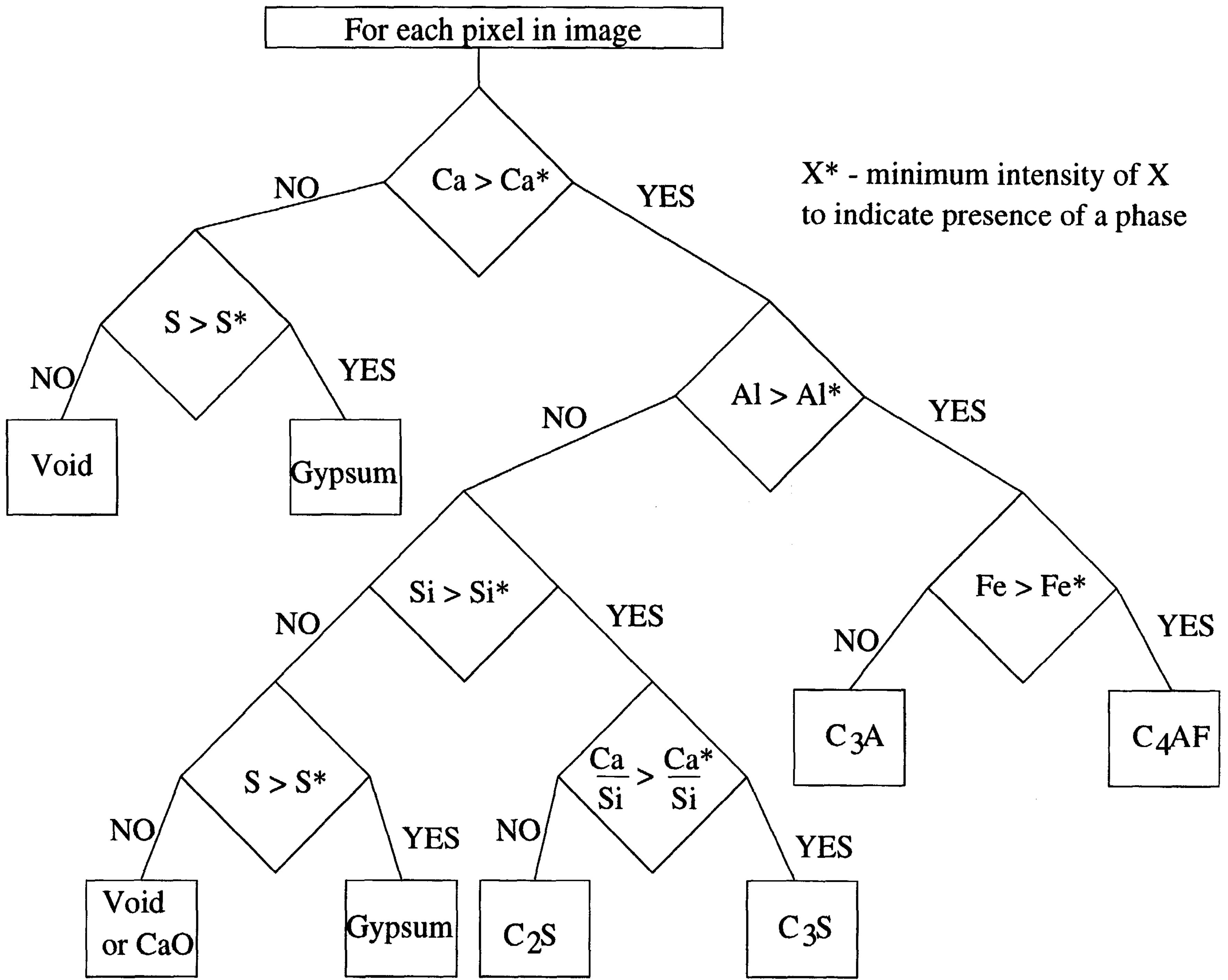


Figure 2: Flowchart specifying segmentation algorithm for selecting pixel phase values.

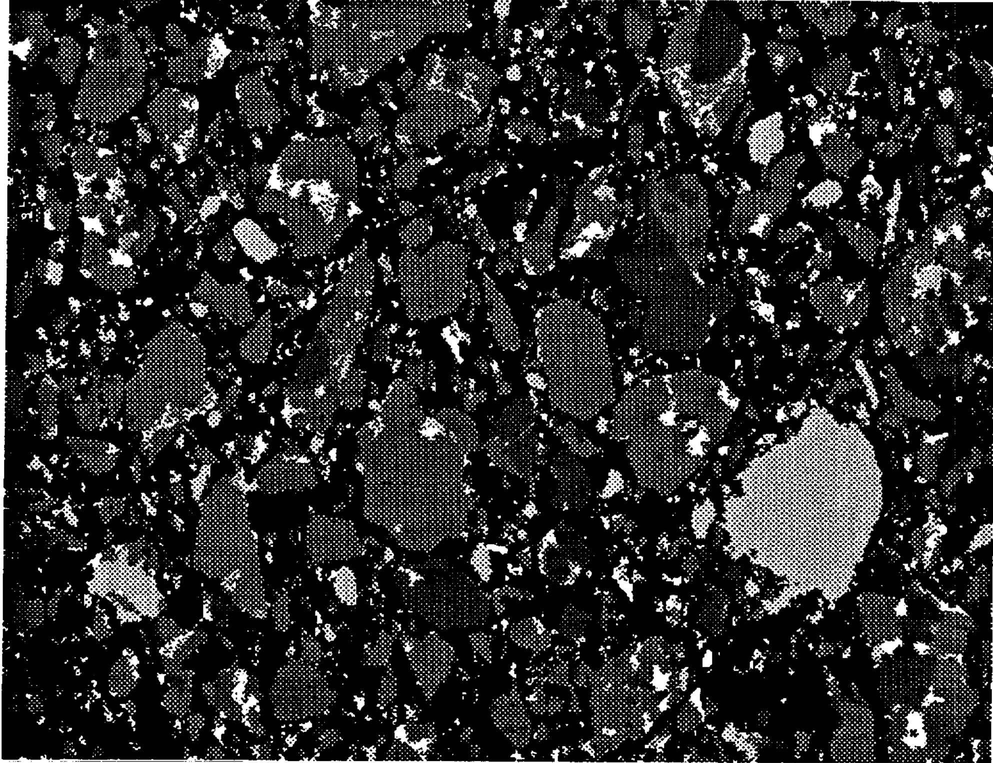


Figure 3: Final segmented two-dimensional image of Cement 116 issued by the Cement and Concrete Reference Laboratory (NIST). Phases from brightest to darkest are: C_3A, gypsum, C_4AF, C_3S, C_2S, and porosity. Image is approximately 250 μm $\times$ 200 μm.

The user must input the size ($M \times N$) of the image and also a computational "mask" to be used in the correlation calculation. The mask is used so that simple **logical and** calculations may be employed in computing the correlation function and consists of the sum of 2 raised to the ID for each phase of interest. For example, to determine the autocorrelation for the two silicate phases (C_3S and C_2S), the mask would be (2^1+2^2) or 6. Similarly, the masks for C_3S, C_3A, and C_4AF would be 2, 8, and 16, respectively. The user can also specify a scaling factor, n (1 or 2 for instance), indicating that only every nth pixel should be considered in the autocorrelation calculation, in case the resolution of the final processed SEM image is greater than the 1 μm/pixel normally employed in the cement hydration model.

Once S(x,y) is determined, it is converted to S(r) format, since for an isotropic media the autocorrelation function should only be a function of distance $r = \sqrt{x^2+y^2}$ and not (x,y) [8]. For this conversion, the program **corrxy2r.c** provided in Appendix A is used. It accepts as input a file output by the program corrcalc and returns a file containing a listing of S(r) vs. r. The following equation is utilized for the conversion from Cartesian to polar coordinates:

$$S(r) = \frac{1}{2r+1} \sum_{l=0}^{2r} S(r, \frac{\pi l}{4r}) \tag{2}$$

where S(r, θ)=S(r cosθ,r sinθ). For non-integer values of (r cosθ) and (r sinθ), S(r, θ) is obtained by bilinear interpolation from the input values of S(x,y), which are only available

for integer values of x and y. The properties of the autocorrelation function are such that S(0) provides the area fraction, P_a, of the phase(s) of interest and as r $\rightarrow \infty$, S(r) $\rightarrow P_a^2$.

3 Two-dimensional to Three-dimensional Conversion

3.1 Generation of spherical particles following measured PSD

The program **genpart3d.c**, whose listing is provided in Appendix B, is used to place digitized spherical particles of a user specified particle size distribution into a three-dimensional computational volume, typically 100 pixels on a side. Initially, the user must provide a negative integer to be used as the random number seed. Following this, the program is menu driven with the following main menu options:

1) Exit: Exit the program

2) Add spherical particles (cement and gypsum) to microstructure: allows the user to specify the discretized particle size distribution that should be used, the number of particles to place, the proportion of particles which should be gypsum as opposed to cement, and whether the particles should be dispersed. The particle size distribution should be on a number basis. Since most particle size analyzers provide the distribution on a mass basis, a mass to number basis conversion (easily implemented in a spreadsheet) may be required. As input, the user must specifically supply the number of different size spheres to use (parameter NUMSIZES determines the maximum number of different sizes allowed), a dispersion factor (0, 1, or 2) which specifies the minimum pixel separation to be maintained between all pairs of particles, and a probability (0.0 to 1.0) for the generation of gypsum particles instead of cement. Following this, the user, for each different size class of spheres, provides the number and radius (size) characterizing that size class. The user should always begin with the largest particles and proceed consecutively to the smallest particles. Otherwise, after placing some of the smallest particles, no spaces where the largest particles can fit may remain in the 3-D microstructure. The dispersion capability has been included to simulate the effects of adding a superplasticizer to the cement paste. However, it should be noted that for lower water-to-cement ratios (< 0.45), it may not be possible to place all of the requested particles in a dispersed configuration. In this case, the program will exit after parameter MAXTRIES unsuccessful attempts at finding a random location for a particle. Table 2 provides a listing of the number of pixels contained in spheres of various diameters in pixels, which should prove useful in creating user specific PSDs. It should be noted that the $diameter = 2 * radius + 1$ so that all spheres may be centered exactly on a pixel.

3) Flocculate system by reducing number of particle clusters: allows the user to create any desired number of flocs (clusters) by randomly moving each particle (cluster) centroid in one-pixel increments and aggregating any particles (clusters) which contact one another during this process. The only input required is the user requested number of clusters (flocs) to be present at the end of execution of the routine. Typically, if no superplasticizer or water reducing agent is used, the cement particles will have a great tendency to flocculate together [9], perhaps into a single floc.

Table 2: Volume in Pixels Occupied by Spheres of Various Diameters

Diameter (pixels)	Pixels per sphere
3	19
5	81
7	179
9	389
11	739
13	1189
15	1791
17	2553
19	3695
21	4945
23	6403
25	8217
27	10395
29	12893
31	15515
33	18853
35	22575
37	26745
39	31103
41	36137

4) Measure phase fractions: outputs the number of pixels of cement, gypsum, and aggregate present in the 3-D microstructure.

5) Add an aggregate to the microstructure: allows the user to add a single flat plate aggregate to the 3-D microstructure for studying the development of microstructure in the interfacial transition zone [10]. The only required input is the thickness of the aggregate to be placed, which must be an even integer. Typically, the user should place an aggregate (if desired) into the microstructure before placing any of the cement particles. Otherwise, the aggregate will simply overlap and replace any cement particle or gypsum pixels contained within its boundaries.

6) Measure single phase connectivity: allows the user to employ a burning algorithm [11] to determine the percolation characteristics of either the porosity or the solids present in the 3-D microstructure. The user must specify the phase in which they are interested and the routine returns the number of pixels of that phase which are accessible from the top of the 3-D microstructure along with the number of pixels which are contained in pathways that traverse (span) the microstructure.

7) Measure phase fractions vs. distance from aggregate: outputs a listing of the number of pixels of each phase (cement, gypsum, porosity) present in parallel planes at various fixed distances (one pixel increments) from the aggregate surface.

8) Output microstructure to file: allows the user to save the created microstructure to files. The user must supply two filenames, one for the storage of the actual microstructure (cement, gypsum, and porosity), and the second for storage of the individual particle IDs (to be used in assessing the setting of the cement during hydration).

3.2 Filtering of random noise particle image

Once a 3-D image incorporating the desired particle size distribution has been created, the next step is to introduce the appropriate phase volume fractions, phase surface area fractions, and correlation structure into the initially monophase particles. This process is accomplished in a series of steps using one Fortran and two C programs. The Fortran program, **rand3d.f** (listing provided in Appendix B), is used to introduce the correct phase volume fraction and the correlation structure measured on the 2-D SEM image into the 3-D cement particle image. Starting with an image of random Gaussian noise, generated using the Box-Muller method [12], the measured autocorrelation function for the phase(s) of interest is used to filter the image, introducing the appropriate correlation structure. Each time this program is executed, the user must specify what phase is to be subdivided into two phases and the value to be assigned to the new phase. A typical sequence is to separate the cement into silicates and aluminates, separate the silicates into C_3S and C_2S, and separate the aluminates into C_3A and C_4AF, as illustrated in Fig. 4.

To use **rand3d.f**, the user must input:

a random number seed (negative integer),

the initial phase ID to be subdivided into two phases,

the phase ID for the second (new) phase,

the name of the file containing the current 3-D microstructure to be processed,

the name of the file containing the correlation data for the phase(s) of interest, and

the phase volume fraction (0.0 to 1.0) of the initial phase which is to retain its original phase value.

The final input, the phase volume fraction, can be determined from the 2-D SEM image or from a conventional Bogue analysis of the cement oxide composition [13], being sure to convert from a mass to a volume basis in the latter case. The program stores the resulting microstructure one integer per line in a file named **IMAGE3D.OUT**, which the user may rename and retain for further processing.

3.3 Correction of hydraulic radius

Because the correlation structure of the 3-D image produced by **rand3d.f** only approximates that of the input 2-D correlation function, a modification is utilized to directly match the surface area fraction (via the hydraulic radius) of the 3-D image to its 2-D counterpart, greatly improving the agreement between the 2-D and 3-D correlations [14]. Typically, the IMAGE3D.OUT microstructure file produced by **rand3d.f** is renamed and analyzed using the program **stat3d.c**. This C program (listing provided in Appendix B), will determine

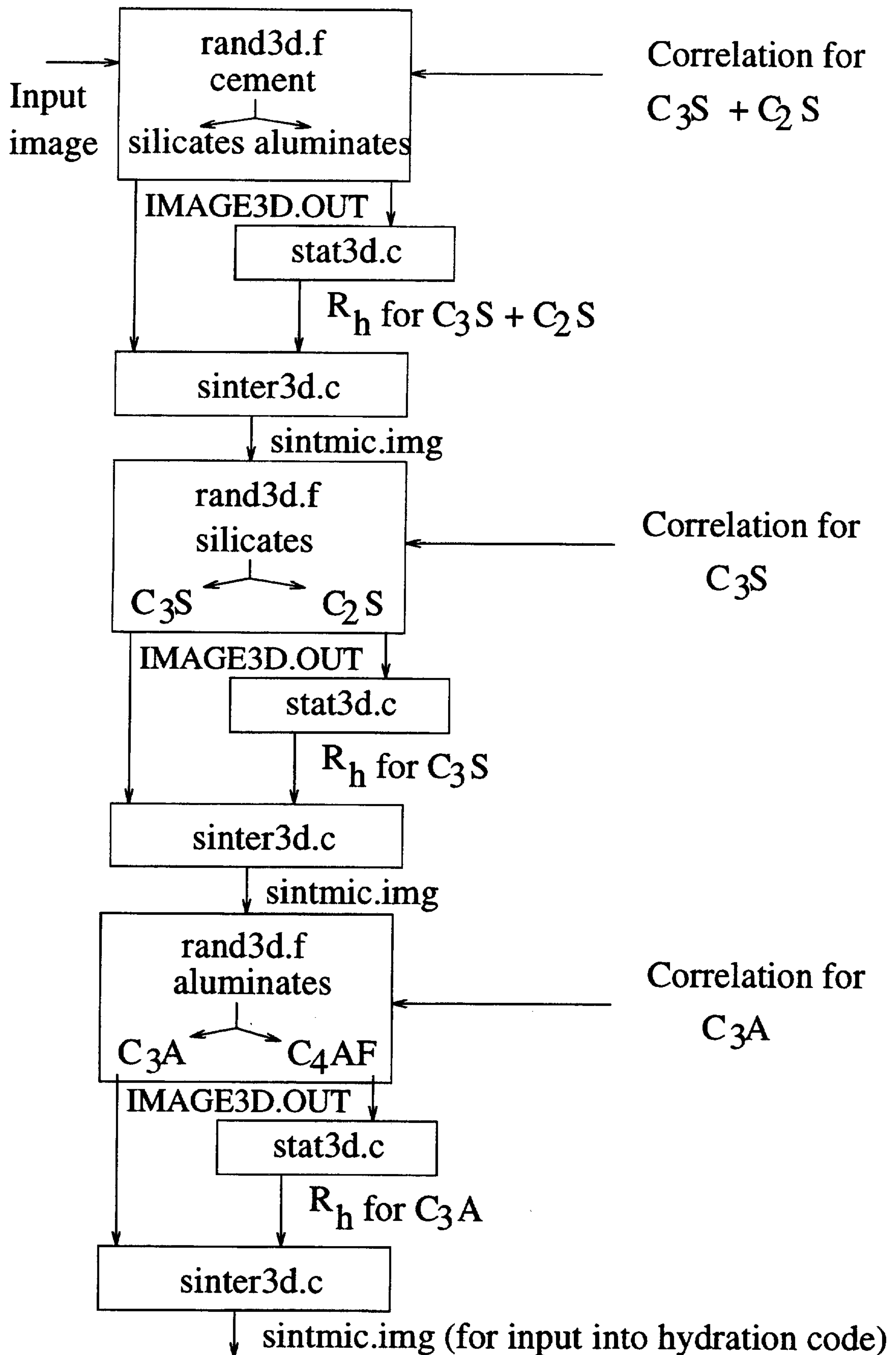


Figure 4: Flowchart specifying filtering algorithm for assigning pixel phase values to the three-dimensional starting cement microstructure image.

the phase volume fractions and surface area fractions for each phase present in the microstructure. The surface area fractions are reported on a gypsum-free basis, so the user must be sure that the 2-D image has also been analyzed on a gypsum-free basis (i.e., considering only the C_3S, C_2S, C_3A, and C_4AF). The needed surface area count for a phase can be determined by multiplying the total surface area count by the surface (perimeter) fraction determined in analyzing the 2-D SEM image. From this, the appropriate value of hydraulic radius, R_h, can be determined as:

$$R_h = \frac{6}{4} * \frac{volume\ in\ pixels}{needed\ surface\ area\ in\ pixels}. \quad (3)$$

The value of 6/4 is included in the equation to correct for the fact that a digitized sphere has a surface area of approximately $6\pi r^2$ as opposed to $4\pi r^2$.

This hydraulic radius is input into the program **sinter3d.c** (listing provided in Appendix B) which interchanges pixels of two specific phases to obtain the specified hydraulic radius for the first phase. This code was originally developed to simulate the sintering of a ceramic powder by exchanging solid pixels of high curvature with porosity pixels of low curvature [15]. The program is menu driven and the user must first read in the microstructure previously output by execution of **rand3d.f**, using menu selection 4. After this, the sintering algorithm (menu selection 2) can be executed to adjust the hydraulic radius. For this algorithm, the following inputs are required:

the two phase IDs between which to execute the sintering algorithm; the first phase indicates the one whose hydraulic radius will be increased,

the number of pixels of each phase to exchange in each cycle of the sintering algorithm (default value=200),

the calculated target hydraulic radius (equation 3), and

the radius of the digitized sphere [15] to be used in assessing curvature (default value=3).

The sintering algorithm will be iteratively executed until the desired hydraulic radius is achieved or an equilibrium is reached. At this point, the user may elect to output the resulting microstructure to a file using menu selection 3. A file named **sintmic.img** will be created for this output. The user may then exit the program and rename this output file to retain it for future use.

4 Three-dimensional Cement Hydration Model

4.1 Inputs

The following inputs are required for execution of the hydration model (program **disreal3d.c** provided in Appendix C):

a negative integer random number seed,

a flag (0=No, 1=Yes) indicating if the final microstructure is to be output to a file. If a value of 1 is entered for this parameter, the next entry must be the filename of the file to be created to store this microstructure.

the filename of the file containing the initial 3-D microstructure to be used in the hydration model,

for this file, the integer values assigned to C_3S, C_2S, C_3A, C_4AF, gypsum, and aggregate (separated by spaces),

the filename of the file containing the initial 3-D particle ID microstructure to be used in the hydration model (for assessing setting behavior),

the number of one pixel particles of a phase to add. The program contains an iterative loop to continue to accept non-zero values for this parameter, so that the user can place different types of one pixel particles, at their discretion. This iterative process is terminated by the user inputting a value of 0 for the number of one pixel particles to add. Every time a non-zero value is input, the next entry must be the phase ID of the particles to be placed. The purpose of this input is to allow the user to add one-pixel inert (ID=13), silica fume (pozzolanic, ID=12), or gypsum (ID=6) particles to an existing 3-D starting microstructure prior to hydration.

the number of cycles of the hydration model to execute,

a flag indicating if hydration is to be under (0) saturated or (1) sealed conditions,

the maximum number of diffusion steps to take in a given dissolution cycle,

a prefactor and a scale factor for the nucleation probability of CH according to an exponential function [1],

a prefactor and a scale factor for the nucleation probability of C_3AH_6,

a prefactor and a scale factor for the nucleation probability of FH_3,

the frequency (in cycles) for examining the percolation properties of the capillary pore space (this examination can be totally avoided by setting this parameter to a value larger than the requested number of cycles),

the frequency (in cycles) for examining the percolation properties of the solids (set point),

the induction time (*time_induct*) in hours for use in converting model cycles to real time,

the initial temperature of the system in degrees Celsius,

the activation energy (kJ/mole) for the cement hydration reactions,

the calibration factor (β) for converting model cycles to real time based on an equation of the form $time = time_induct + \beta \times cycles \times cycles$ [1],

the mass fraction of aggregates (0.0 for cement paste hydration) in the concrete mixture proportions (not that present in the 3-D microstructure being used but that present in the concrete mixture being simulated), and

a flag indicating if hydration is to be under (0) isothermal or (1) adiabatic conditions.

Several example input datafiles are provided in Appendix D.

4.2 Model execution

The 3-D cement hydration and microstructure development model is described in detail elsewhere [1, 2]. Once the initial 3-D microstructure and particle ID structures are read in, hydration is executed as a series of dissolution/diffusion/reaction cycles. In dissolution, any pixels of a soluble phase which are in contact with capillary pore space may dissolve and enter solution as one or more diffusing species. These diffusing species then undergo random walks within the pore space until a reaction occurs. After a specified number (typically 500) of diffusion steps have been taken, phase counts are tabulated, empty porosity created to account for the chemical shrinkage (if hydration is under sealed conditions), and another cycle of dissolution is executed. All diffusing species IDs and locations are stored in a dynamically allocated doubly linked list, so that the diffusing species may remain in solution from one dissolution cycle to the next. However, during the last diffusion step of the last cycle of the hydration, all diffusing species are converted into either the phase from which they dissolved or the appropriate hydration products.

In addition, after each dissolution cycle, the heat released by the reactions (as described in the next section) and the degree of hydration which has occurred are used to update the heat capacity and temperature of the system. The heat capacity is a function of degree of hydration, and undergoes about a 10% decrease during the course of hydration due to changes in the state of the water present in the cement paste [20]. Dividing the incremental heat release by the current heat capacity determines the incremental temperature rise of the system. In this way, the adiabatic response of a concrete system may be predicted by estimating temperature as a function of equivalent real time, calculated based on the application of the principles of the maturity method [21] as described in Section 5.3.

4.3 Outputs

In addition to the capability of writing the final hydrated microstructure to a file, four additional files are created during the execution of **disreal3d**. The first of these contains a log of the program execution and is written to stdout (standard output). Thus, to save this to file, the user would typically pipe the output, perhaps using a command line of the form:

```
disreal3d <disreal3d.dat >disreal3d.out.
```

This file contains a listing of all user inputs, the number of diffusing species created during each dissolution, the number of pixels of each phase present during each cycle, and information on the assessment of percolation properties of the pore space and the total solids.

The second file, **phases.out**, contains for each cycle the number of pixels of each phase present at the end of execution of the cycle and the number of pixels of water remaining in the system (for hydration under sealed or self-desiccating conditions). These phase counts are given in the order in which the phases are listed (0-22) in the **disreal3d.c** program provided in Appendix C.

The third file, **heat.out**, contains for each cycle, the degree of hydration achieved both on a volume and a mass basis and the estimated heat release using four different methods, the first based on the specific enthalpy values of each phase and the latter three based on different values for the heats of hydration of each of the four major clinker phases, as documented in the code listing provided in Appendix C. It is the fourth value that is currently utilized in calculating the temperature rise for hydration under adiabatic conditions.

The fourth file, **adiabatic.out**, contains an estimate of the adiabatic heat signature for a concrete or mortar with the 3-D cement paste as its binder component. Here for each cycle, the file contains the estimated equivalent time in hours, the temperature in degrees Celsius, the degree of hydration on a mass basis, the estimated reaction rate at the current temperature, the current estimate of the heat capacity of the mortar or concrete in J/g/°C, and the current mass fraction of cement in the system. This latter value should remain constant unless the hydration is being executed under saturated conditions, in which case the additional water imbibed into the hydrating cement paste will reduce the mass fraction of cement in the overall system.

The latter three of these files are in a format which can be easily imported into a spreadsheet or read directly into a plotting package. This allows the user to compare results from different cements, w/c ratios, or hydration conditions or to produce plots of various properties vs. the number of hydration cycles or the estimated equivalent real time for comparison to experimental data.

5 Example Applications

5.1 Set Point- Percolation of Solids

In the first application, the datafile for which is provided in Appendix D.2, we will examine the percolation of total solids, related to setting behavior, of a cement paste with a w/c ratio of 0.4, for a system in which all of the cement particles have been flocculated into a single floc structure. For this application, the hydration will be executed under isothermal and saturated conditions. In addition, the dissolution bias parameter in the program **disreal3d.c**, DISBIAS, has been increased from 20.0 to 100.0, to reduce the dissolution rate and increase our resolution in observing the percolation behavior. Twenty cycles of the hydration are executed, with the solids percolation being determined after each cycle. In the model, total solids percolation is defined as the fraction of cement particle and hydration product (specifically $C–S–H$ and ettringite) pixels for which the cement particles are connected together by either ettringite or $C–S–H$ gel. Thus, two touching cement particles do not comprise a connected pathway unless they are also bridged by either a layer of $C–S–H$ or ettringite.

Figure 5 provides a plot of the connected fraction of total solids vs. the achieved degree of hydration for this system. In this case, set occurs in the range of hydration of 2.5 to 3%, consistent with previous computer modelling results [16] using a model based solely on C_3S hydration [17] and experimental results based on measuring a shear resistance of 0.08 MPa for the cement paste [9].

5.2 Percolation Threshold of Capillary Pore Space

Another percolation property of interest in cement-based materials is the percolation threshold of the capillary pore space as a function of hydration [18]. As hydration proceeds and capillary porosity is reduced, a point may be reached where the capillary porosity is no longer connected in three dimensions, exemplifying a percolation transition. As an example, we can consider the fraction of the total capillary porosity which remains connected across the 3-D microstructure as a function of the volume fraction of porosity for the same w/c=0.4 cement paste system considered in the previous example. The datafile used for this execution is provided in Appendix D.3. In this example, the dissolution bias parameter remains at its

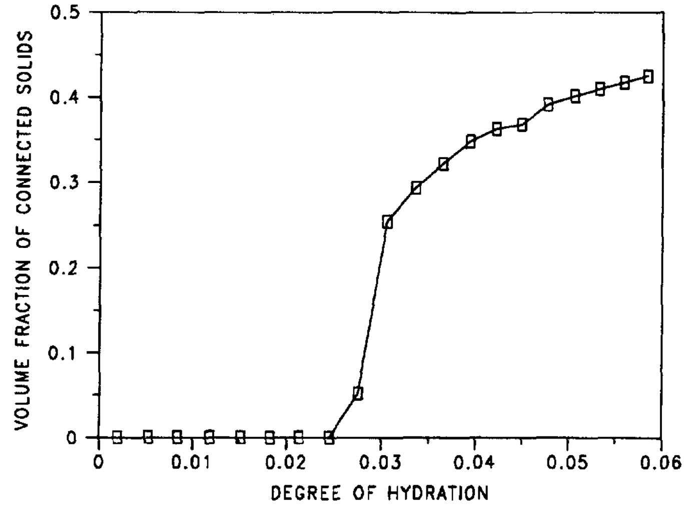


Figure 5: Connected total solids volume fraction vs. degree of hydration for a cement paste with a w/c ratio of 0.4.

"default" value of 20.0 as we are interested in examining this percolation over a broad range of hydration. Figure 6 provides a plot of the fraction of connected capillary porosity vs. the total volume fraction of capillary porosity during the hydration. The capillary porosity is seen to depercolate at about 20% capillary porosity, in agreement with previous computer modelling [18] and experimental [19] results.

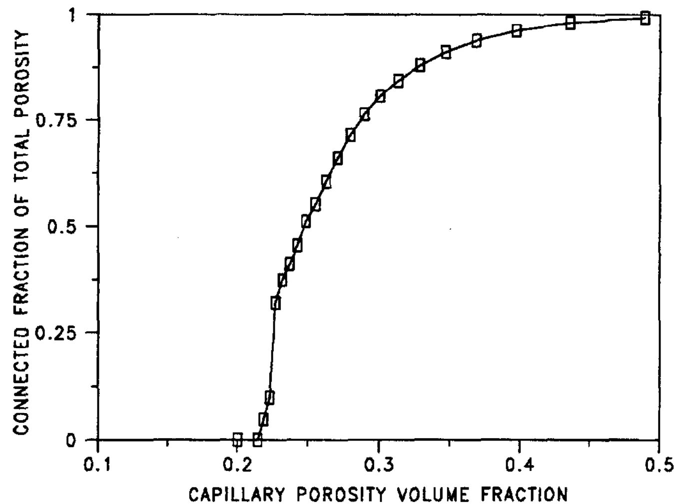


Figure 6: Connected capillary porosity fraction vs. volume fraction of porosity for a cement paste with a w/c ratio of 0.4.

5.3 Prediction of Adiabatic Heat Signature

Recently, the 3-D cement hydration and microstructural development model has been modified to include the capability to predict the adiabatic temperature rise for a concrete, i.e.

the temperature vs. time curve which would be obtained for a concrete maintained under adiabatic (no heat lost to the surrounding environment) conditions. As hydration occurs, the cumulative heat generated by the hydration reactions is tabulated and the heat capacity of the concrete updated. Based on these two values, the incremental temperature rise from one cycle of dissolution to the next can be calculated as:

$$\Delta T = \frac{[H(i) - H(i-1)]}{C_p(i)} \quad (4)$$

where ΔT is the incremental change in temperature from cycle (i-1) to cycle i, $C_p(i)$ is the value of the heat capacity for the concrete at cycle i, and H(i) is the cumulative heat generated by the hydration reactions through cycle i. In addition, the equivalent real time which elapses during this cycle of hydration is estimated using principles based on the maturity method [2, 21] and a user-supplied calibration factor relating cycles to equivalent time at a temperature of 25 °C.

Previously [1, 2], the following conversion between cycles and time has been employed to calibrate model results to experimental data:

$$t = t_0 + \beta * cycles * cycles \quad (5)$$

where t_0 is a user-supplied induction time for the cement of interest. The differential time elapsing between cycles (i-1) and i can be determined by substitution into equation 5 and subsequent subtraction to obtain:

$$\Delta t(i) = (2. * i - 1) * \beta. \quad (6)$$

This increment in time would correspond to a constant temperature of reaction (typically 25 °C). To adjust this value for the changing temperature under adiabatic curing conditions, the maturity method is adapted [2]. The variation in reaction rate, k, with temperature can be described by an Arrhenius type relationship [21]:

$$k_T = k_0 * e^{\frac{-E_a}{RT}} \quad (7)$$

where E_a is an apparent activation energy, R is the universal gas constant (8.314 J/(mole K)), and T is absolute temperature in K. An equivalent time, t_e, at any temperature of interest, T_i, relative to a base reference temperature, T_r, can be calculated as:

$$t_e = \frac{k_T}{k_r} * t = e^{-\frac{E_a}{R} * (\frac{1}{T_i} - \frac{1}{T_r})} * t \quad (8)$$

where k_T is the rate constant at the experimental temperature of interest and k_r is the rate constant at the reference temperature. Finally, by combining equations 6 and 8, one obtains:

$$\Delta t_e(i) = (2 * i - 1) * \beta * \frac{k_T}{k_r}. \quad (9)$$

Figure 7 shows an example comparison between experimental and model predicted temperature rise, based on executing the program **disreal3d** with the input datafile provided in Appendix D.1. The agreement between model and experiment is quite good, particularly for times less than 30 hours, but it should be kept in mind that the user has had to supply an induction time, initial temperature, activation energy, and cycle-time conversion factor as input parameters to the program.

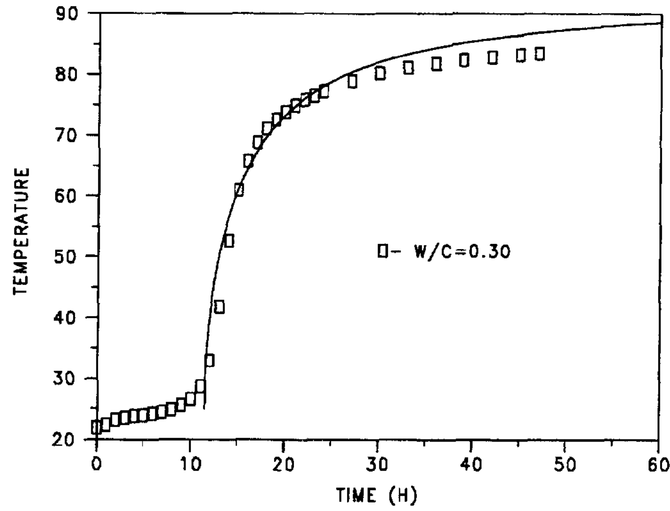


Figure 7: Experimentally measured and model predicted (solid line) adiabatic temperature rise for an ordinary portland cement concrete containing 72% aggregates on a mass basis.

6 Acknowledgements

The author would like to thank Paul Stutzman of BFRL/NIST for providing the source SEM and X-ray images used in reconstructing the three-dimensional cement particles and Vincent Waller and Francois DeLarrard of LCPC, Paris, France for providing the experimental data and the experimentally measured activation energy for the model calculation in Figure 7.

7 References

[1] Bentz, D.P., *A Three-Dimensional Cement Hydration and Microstructure Program. I. Hydration Rate, Heat of Hydration, and Chemical Shrinkage*, NISTIR **5756**, U.S. Department of Commerce, November 1995 (available from National Technical Information Service).

[2] Bentz, D.P., *Journal of the American Ceramic Society*, **80** (1), 3-21, 1997.

[3] Bentz, D.P., Clifton, J.R., and Snyder, K.A., *Concrete International*, **18** (12), 42-47, 1996.

[4] Struble, L.J. and Stutzman, P.E., *Journal of Materials Science Letters*, **8**, 632-634, 1989.

[5] Stutzman, P.E., "Applications of Scanning Electron Microscopy in Cement and Concrete Petrography," Petrography of Cementitious Materials, ASTM STP 1215, edited by Sharon M. DeHayes and David Stark (American Society for Testing and Materials, Philadelphia, 1994), pp. 74-90.

[6] Bentz, D.P. and Stutzman, P.E., "SEM Analysis and Computer Modelling of Hydration of Portland Cement Particles," Petrography of Cementitious Materials, ASTM STP 1215, edited by Sharon M. DeHayes and David Stark (American Society for Testing and Materials, Philadelphia, 1994), pp. 60-73.

[7] Castleman, K.R., Digital Image Processing (Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979).

[8] Berryman, J.G., *Journal of Applied Physics*, 57 (7), 2374-2384, 1985.

[9] Jiang, S.P., Mutin, J.C., and Nonat, A., *Cement and Concrete Research*, **25** (4), 779-789, 1995.

[10] Bentz, D.P., Schlangen, E., and Garboczi, E.J., "Computer Simulation of Interfacial Zone Microstructure and Its Effect on the Properties of Cement-Based Composites," Materials Science of Concrete IV, edited by Jan P. Skalny and Sidney Mindess (American Ceramic Society, Westerville, OH, 1995), pp. 155-200.

[11] Stauffer, D., Introduction to Percolation Theory (Taylor and Francis, London, 1985).

[12] Law, A.M, and Kelton, W.D., Simulation Modeling and Analysis (McGraw-Hill, New York, 1982).

[13] Taylor, H.F.W., Cement Chemistry (Academic Press, London, 1990).

[14] Bentz, D.P. and Martys, N.S., *Transport in Porous Media*, **17** (3), 221-238, 1995.

[15] Bullard, J.W., Garboczi, E.J., Carter, W.C., and Fuller, E.R., *Computational Materials Science*, **4**, 1-14, 1995.

[16] Bentz, D.P., Garboczi, E.J., and Martys, N.S., "Application of Digital-Image-Based Models to Microstructure, Transport Properties, and Degradation of Cement-Based Materials," Computer Modelling of Microstructure and Its Potential Application to Transport Properties, edited by H.M. Jennings and J. Kropp (Kluwer Academic Pulishers, The Netherlands, 1995), pp. 167-185.

[17] Bentz, D.P. and Garboczi, E.J., *Guide to Using HYDRA3D: A Three-Dimensional Digital-Image-Based Cement Microstructural Model*, NISTIR **4746**, U.S. Department of Commerce, January 1992.

[18] Bentz, D.P. and Garboczi, E.J., *Cement and Concrete Research*, **21**, 325-344, 1991.

[19] Powers, T.C., Copeland, L.E., and Mann, H.M., *PCA Bulletin*, **10**, 1959.

[20] Waller, V., DeLarrard, F., and Roussel, P., "Modelling the Temperature Rise in Massive HPC Structures," *4th International Symposium on Utilization of High-Strength/High-Performance Concrete*, Paris, Paper 170, 415-421, 1996.

[21] Carino, N.J., Knab, L.I., and Clifton, J.R., *Applicability of the Maturity Method to High-Performance Concrete*, NISTIR **4819**, U.S. Department of Commerce, May 1992.

Appendices

A Computer programs for two-dimensional imaging

A.1 Listing for statsimp.c

```
/************************************************************************/
/*                                                                      */
/*      Program: statsimp.c                                             */
/*      Purpose: To compute the area and surface areas                  */
/*              for one or more phases of a cement particle image       */
/*      Programmer: Dale P. Bentz                                       */
/*                  NIST                                                */
/*                  Building 226 Room B-350                             */
/*                  Gaithersburg, MD  20899-0001                        */
/*                  Phone: (301) 975-5865                               */
/*                  E-mail: dale.bentz@nist.gov                         */
/*                                                                      */
/************************************************************************/
#include <stdio.h>
#include <math.h>

int flag[5];

/* Note that program is limited to a 512*512 image size */
main()
{
        int mask,i,j,image [512] [512],xsize,ysize;
        int inval;
        long int sum,edgesum;
        float fsum;
        char filen[80];
        FILE *infile;

        do{
                printf("Enter size of image in x and y directions \n");
                scanf("%d %d",&xsize,&ysize);
                printf("%d %d \n",xsize,ysize);
        } while ((xsize>512)||(ysize>512));

        printf("Enter name of image file \n");
        scanf("%s",filen);

        infile=fopen(filen,"r");
```

```
	mask=1;
	/* Read in the image and assign powers of two as phase values */
	for(i=0;i<xsize;i++){
	for(j=0;j<ysize;j++){
		fscanf(infile,"%d\n",&inval);
		image [i] [j]=(int)(pow(2.,(float)inval));
	}
	}
	fclose(infile);

/* Now perform calculation */
/* Allow user to execute counting as many times as desired */
	while(mask!=0){
		printf("Phase assignments are assumed to be as follows: \n\n");
		printf("Phase      Image ID   Mask value \n \n");
		printf("Pores         0            1\n");
		printf("C3S           1            2\n");
		printf("C2S           2            4\n");
		printf("C3A           3            8\n");
		printf("C4AF          4           16\n");

	   printf("Enter composite value for mask to use during this run \n");
	   printf("Example: for C3S and C2S, composite would be 2+4=6 \n");
		printf("Enter 0 to exit program \n");
		scanf("%d",&mask);
		printf("%d\n",mask);

		/* Determine area and perimeter counts for this phase(s) */
		/* Perimeter count is number of pixel edges of phase(s) */
		/* in contact with porosity */
		/* Non-periodic so ignore a one-pixel layer around the edge */
		if(mask!=0){
		sum=edgesum=0;
		for(i=1;i<(xsize-1);i++){
		for(j=1;j<(ysize-1);j++){

		if((mask&(image[i][j]))!=0){
			sum+=1;
			/* Check immediate 4 neighbors for edges */
			if(((image[i-1][j]))==1){
				edgesum+=1;
			}
			if(((image[i+1][j]))==1){
				edgesum+=1;
			}
			if(((image[i][j-1]))==1){
				edgesum+=1;
```

```
                        }
                        if(((image[i][j+1]))==1){
                                edgesum+=1;
                        }
                }

                }
                }

                printf("Area- %ld pixels Perimeter- %ld \n",sum,edgesum);
                fsum=(float)sum/((float)(xsize-2)*(float)(ysize-2));
                printf("Phase fraction is %f \n",fsum);
                }
        } /* end of while mask loop */
}
```

A.2 Listing for corrcalc.c

```
/************************************************************************/
/*                                                                      */
/*      Program: corrcalc.c                                             */
/*      Purpose: To compute the 2-D correlation function in S(x,y) form */
/*              for one or more phases of a cement particle image       */
/*      Programmer: Dale P. Bentz                                       */
/*                  NIST                                                */
/*                  Building 226 Room B-350                             */
/*                  Gaithersburg, MD  20899-0001                        */
/*                  Phone: (301) 975-5865                               */
/*                  E-mail: dale.bentz@nist.gov                         */
/*                                                                      */
/************************************************************************/
#include <stdio.h>
#include <math.h>

#define CSIZE 60   /* Extent of correlation computation in pixels */

int flag[5];

/* Note that maximum image size is 512*512 */
main()
{
        int mask,i,j,image [512] [512],xsize,ysize;
        int scalef,xoff,yoff,inval,iscale,jscale;
        long int sum,ntot;
        float fsum;
        char filen[80],fileo[80];
```

```
        FILE *infile,*outfile;

        do{
                printf("Enter size of image in x and y directions \n");
                scanf("%d %d",&xsize,&ysize);
                printf("%d %d \n",xsize,ysize);
        } while ((xsize>512)||(ysize>512));

/* Assumed file format is a simple list of integers (one per line) */
/* representing phases as indicated below and with y in the inner loop */
        printf("Enter name of image file \n");
        scanf("%s",filen);
        infile=fopen(filen,"r");

        printf("Enter name of correlation file to create\n");
        scanf("%s",fileo);
        outfile=fopen(fileo,"w");
/* Output header consisting of the extent of the correlation matrix */
        fprintf(outfile,"%d %d\n",CSIZE+1,CSIZE+1);

/* If original image is acquired at a resolution much less than */
/* 1 um/pixel, scaling factor can be used to for example skip every other */
/* pixel to adjust to appropriate resolution */
        scalef=1;
        printf("Enter scaling factor to use (default=%d) \n",scalef);
        scanf("%d",&scalef);
        printf("%d \n",scalef);

        printf("Phase assignments are assumed to be as follows: \n\n");
        printf("Phase       Image ID   Mask value \n");
        printf(" \n");
        printf("Pores, etc.    0           1\n");
        printf("C3S            1           2\n");
        printf("C2S            2           4\n");
        printf("C3A            3           8\n");
        printf("C4AF           4          16\n");

/* Composite mask is a simple logical and of the above values */
/* Example: For C3S+C2S, mask would be 2&4 = 6 */
        printf("Enter composite value for mask to employ during this run \n");
printf("Example- for C3S+C2S, composite value would be 2+4=6 \n");
        scanf("%d",&mask);
        printf("%d\n",mask);

/* Read in the original image and convert values to 2**i format */
/* so that simple and operations may be employed to compute correlation */
        for(i=0;i<xsize;i++){
```

```
        for(j=0;j<ysize;j++){
                fscanf(infile,"%d\n",&inval);
                image [i] [j]=(int)(pow(2.,(float)inval));
        }
        }

        fclose(infile);
        printf("Finished reading in file \n");
        fflush(stdout);

/* Now perform calculation */
/* Correlation is computed only over a distance of CSIZE pixels */
        for(i=0;i<=CSIZE;i++){
        for(j=0;j<=CSIZE;j++){
                sum=0;
                ntot=0;

/* iscale and jscale represent limits of correlation calculation since */
/* original image is assumed to be non-periodic */
                iscale=i*scalef;
                jscale=j*scalef;
/* Be sure to skip to every scalef pixel to adjust resolution */
                for(xoff=0;xoff<(xsize-iscale);xoff+=scalef){
                for(yoff=0;yoff<(ysize-jscale);yoff+=scalef){
                        ntot+=1;
/* Increment counter if both pixels contain one of the necessary phases */
                if(((mask&(image[xoff][yoff]))!=0)&&
                        ((mask&(image[xoff+iscale][yoff+jscale]))!=0)){
                        sum+=1;
                }

                }
                }
/* Determine and output value for S(x,y) */
        fsum=(float)sum/(float)ntot;
        fprintf(outfile,"%d %d %f \n",i,j,fsum);
        fflush(outfile);

        }
        }

        fclose(outfile);
}
```

A.3 Listing for corrxy2r.c

```
/***************************************************************************/
```

```
/*                                                                      */
/*      Program: corrxy2r.c                                             */
/*      Purpose: To convert the 2-D correlation function from S(x,y)    */
/*               form to S(r) form for use in 3-D reconstructions       */
/*      Programmer: Dale P. Bentz                                       */
/*                  NIST                                                */
/*                  Building 226 Room B-350                             */
/*                  Gaithersburg, MD  20899-0001                        */
/*                  Phone: (301) 975-5865                               */
/*                  E-mail: dale.bentz@nist.gov                         */
/*                                                                      */
/************************************************************************/
#include <stdio.h>
#include <math.h>

#define PIVAL 3.1415926

/* Reference for S(x,y)---> S(r) conversion */
/* Berryman, J.G., "Measurement of Spatial Correlation Functions */
/* Using Image Processing Techniques," J. Appl. Phys., 57 (7), 2374-2384,*/
/* 1985 */

main (){
        FILE *infile,*outfile;
        int nx,ny,i,j,x,y,r,l,xm,ym;
        float sorg [106] [106],ssum,snew,z,theta;
        float xt,yt,s1,s2,st,xr;
        char fname [40], wname[40];

        printf("Enter filename containing raw [S(x,y)] data \n");
        scanf("%s",fname);

        infile=fopen(fname,"r");

/* Read in extent of original S(x,y) calculation, usually 60x60 */
        fscanf(infile,"%d %d\n",&nx,&ny);

/* Read in all of the S(x,y) values */
        for(i=0;i<nx;i++){
        for(j=0;j<ny;j++){
                fscanf(infile,"%d %d %f \n",&x,&y,&z);
                sorg [x] [y]=z;
        }
        }

        fclose(infile);
```

```
        printf("Enter filename to write results to \n");
        scanf("%s",wname);

/* Open output file and write out correlation extent as first value */
        outfile=fopen(wname,"w");
        fprintf(outfile,"%d\n",nx-1);

/* Now convert S(x,y) to S(r) format */
        for(r=0;r<(ny-1);r++){

                ssum=0.0;
                xr=(float)r;

                for(l=0;l<=(2*r);l++){

                        if(xr==0){theta=0;}
                        else{theta=PIVAL*(float)l/(4.*xr);}
                        xt=xr*cos(theta);
                        yt=xr*sin(theta);
                        xm=(int)xt;
                        ym=(int)yt;

/* Use bilinear interpolation */
    s1=(sorg [xm] [ym])-(sorg [xm] [ym] - sorg [xm+1] [ym])*(xt-(float)xm);
    s2=(sorg [xm] [ym+1])-(sorg [xm] [ym+1] -sorg [xm+1] [ym+1])*(xt-(float)xm);

                        st=s1-(s1-s2)*(yt-(float)ym);
                        ssum+=st;
                }

                snew=ssum/(2.*(float)r+1.);
                printf("%d %f \n",r,snew);
                fprintf(outfile,"%d %f\n",r,snew);

        }
        fclose(outfile);
}
```

B Computer programs for two-dimensional to three-dimensional conversion

B.1 Listing for genpart3d.c

```
/**************************************************************************/
/*                                                                        */
/*      Program genpart3d.c to generate three-dimensional cement          */
/*              particles in a 3-D box with periodic boundaries.          */
/*      Particles are composed of either cement clinker or gypsum,        */
/*              follow a user-specified size distribution, and can        */
/*              be either flocculated, random, or dispersed.              */
/*      Programmer: Dale P. Bentz                                         */
/*                  Building and Fire Research Laboratory                 */
/*                  NIST                                                  */
/*                  Building 226 Room B-350                               */
/*                  Gaithersburg, MD  20899   USA                         */
/*                  (301) 975-5865      FAX: 301-990-6891                 */
/*                  E-mail: dale.bentz@nist.gov                           */
/*                                                                        */
/**************************************************************************/
#include <stdio.h>
#include <math.h>

#define SYSSIZE 100     /* system size in pixels per dimension */
#define MAXTRIES 15000 /* maximum number of random tries for sphere placement */

/* phase identifiers */
#define POROSITY 0
/* Note that each particle must have a separate ID to allow for flocculation */
#define CEM 100         /* and greater */
#define CEMID 1         /* phase identifier for cement */
#define GYPID 5         /* phase identifier for gypsum */
#define AGG 6           /* phase identifier for flat aggregate */
#define NPARTC 10000    /* maximum number of particles allowed in box*/
#define BURNT 12000     /* this value must be at least 100 > NPARTC */
#define NUMSIZES 20     /* maximum number of different particle sizes */

/* data structure for clusters to be used in flocculation */
struct cluster{
        int partid; /* index for particle */
        int clustid;   /* ID for cluster to which this particle belongs */
        int partphase;  /* phase identifier for this particle (CEMID or GYPID)*/
        int x,y,z,r; /* particle centroid and radius in pixels */
        struct cluster *nextpart;  /* pointer to next particle in cluster */
};
```

```
/* 3-D particle structure (each particle has own ID) stored in array cement */
/* 3-D microstructure is stored in 3-D array cemreal */
static unsigned short int cement [SYSSIZE+1] [SYSSIZE+1] [SYSSIZE+1];
static unsigned short int cemreal [SYSSIZE+1] [SYSSIZE+1] [SYSSIZE+1];
int npart,aggsize;     /* global number of particles and size of aggregate */
int *seed;      /* random number seed- global */
int dispdist;   /* dispersion distance in pixels */
int clusleft;   /* number of clusters in system */
float probgyp; /* probability of gypsum particle instead of cement */
struct cluster *clust[NPARTC]; /* limit of NPARTC particles/clusters */

/* Random number generator ran1 from Computers in Physics */
/* Volume 6 No. 5, 1992, 522-524, Press and Teukolsky */
/* To generate real random numbers 0.0-1.0 */
/* Should be seeded with a negative integer */
#define IA 16807
#define IM 2147483647
#define IQ 127773
#define IR 2836
#define NTAB 32
#define EPS (1.2E-07)
#define MAX(a,b) (a>b)?a:b
#define MIN(a,b) (a<b)?a:b

double ran1(idum)
int *idum;
/* Calls: no routines */
/* Called by: gsphere,makefloc */
{
        int j,k;
        static int iv[NTAB],iy=0;
        void nrerror();
        static double NDIV = 1.0/(1.0+(IM-1.0)/NTAB);
        static double RNMX = (1.0-EPS);
        static double AM = (1.0/IM);

        if ((*idum <= 0) || (iy == 0)) {
                *idum = MAX(-*idum,*idum);
                for(j=NTAB+7;j>=0;j--) {
                        k = *idum/IQ;
                        *idum = IA*(*idum-k*IQ)-IR*k;
                        if(*idum < 0) *idum += IM;
                        if(j < NTAB) iv[j] = *idum;
                }
                iy = iv[0];
        }
        k = *idum/IQ;
```

```
        *idum = IA*(*idum-k*IQ)-IR*k;
        if(*idum<0) *idum += IM;
        j = iy*NDIV;
        iy = iv[j];
        iv[j] = *idum;
        return MIN(AM*iy,RNMX);
}
#undef IA
#undef IM
#undef IQ
#undef IR
#undef NTAB
#undef EPS
#undef MAX
#undef MIN

/* routine to add a flat plate aggregate in the microstructure */
void addagg()
/* Calls: no other routines */
/* Called by: main program */
{
        int ix,iy,iz;
        int agglo,agghi;

/* Be sure aggregate size is an even integer */
        do{
             printf("Enter thickness of aggregate to place (even integer) \n");
             scanf("%d",&aggsize);
             printf("%d\n",aggsize);
        } while ((aggsize%2)!=0);

        if(aggsize!=0){
        agglo=(SYSSIZE/2)-((aggsize-2)/2);
        agghi=(SYSSIZE/2)+(aggsize/2);

        /* Aggregate is placed in yz plane */
        for(ix=agglo;ix<=agghi;ix++){
        for(iy=1;iy<=SYSSIZE;iy++){
        for(iz=1;iz<=SYSSIZE;iz++){

        /* Mark aggregate into both particle and microstructure images */
                cement [ix][iy][iz]=AGG;
                cemreal [ix][iy][iz]=AGG;
        }
        }
        }
        }
```

```
}

/* routine to check or perform placement of sphere of ID phasein */
/* centered at location (xin,yin,zin) of radius radd */
/* wflg=1 check for fit of sphere */
/* wflg=2 place the sphere */
/* phasein and phase2 are phases to assign to cement and cemreal images resp. */
int chksph(xin,yin,zin,radd,wflg,phasein,phase2)
        int xin,yin,zin,radd,wflg,phasein,phase2;
/* Calls: no other routines */
/* Called by: gsphere */
{
        int nofits,xp,yp,zp,i,j,k;
        float dist,xdist,ydist,zdist,ftmp;

        nofits=0;       /* Flag indicating if placement is possible */

/* Check all pixels within the digitized sphere volume */
        for(i=xin-radd;((i<=xin+radd)&&(nofits==0));i++){
                xp=i;
                /* use periodic boundary conditions for sphere placement */
                if(xp<1) {xp+=SYSSIZE;}
                else if(xp>SYSSIZE) {xp-=SYSSIZE;}
                ftmp=(float)(i-xin);
                xdist=ftmp*ftmp;
        for(j=yin-radd;((j<=yin+radd)&&(nofits==0));j++){
                yp=j;
                /* use periodic boundary conditions for sphere placement */
                if(yp<1) {yp+=SYSSIZE;}
                else if(yp>SYSSIZE) {yp-=SYSSIZE;}
                ftmp=(float)(j-yin);
                ydist=ftmp*ftmp;
        for(k=zin-radd;((k<=zin+radd)&&(nofits==0));k++){
                zp=k;
                /* use periodic boundary conditions for sphere placement */
                if(zp<1) {zp+=SYSSIZE;}
                else if(zp>SYSSIZE) {zp-=SYSSIZE;}
                ftmp=(float)(k-zin);
                zdist=ftmp*ftmp;

        /* Compute distance from center of sphere to this pixel */
                dist=sqrt(xdist+ydist+zdist);
                if((dist-0.5)<=(float)radd){
                        /* Perform placement */
                        if(wflg==2){
                                cement [xp] [yp] [zp]=phasein;
                                cemreal [xp][yp][zp]=phase2;
```

```
                        }
                        /* or check placement */
                        else if((wflg==1)&&(cement [xp] [yp] [zp] !=POROSITY)){
                                nofits=1;
                        }
                }
                /* Check for overlap with aggregate */
        if((wflg==1)&&((abs(xp-((float)(SYSSIZE+1)/2.0)))<((float)aggsize/2.0))){
                        nofits=1;
                }
        }
        }
        }

        /* return flag indicating if sphere will fit */
        return(nofits);
}

/* routine to place spheres of various sizes and phases at random */
/* locations in 3-D microstructure */
/* numgen is number of different size spheres to place */
/* numeach holds the number of each size class */
/* sizeeach holds the radius of each size class */
void gsphere(numgen,numeach,sizeeach)
        int numgen;
        long int numeach[NUMSIZES];
        int sizeeach[NUMSIZES];
/* Calls: makesph, ran1 */
/* Called by: create */
{
        int count,x,y,z,radius,ig,tries;
        long int jg;
        float rx,ry,rz,testgyp;
        struct cluster *partnew;

/* Generate spheres of each size class in turn (largest first) */
        for(ig=0;ig<numgen;ig++){

                radius=sizeeach[ig];     /* radius for this class */
                /* loop for each sphere in this size class */
                for(jg=1;jg<=numeach[ig];jg++){

                        tries=0;
                        /* Stop after MAXTRIES random tries */
                        do{
                                tries+=1;
                        /* generate a random center location for the sphere */
```

```
                                x=(int)((float)SYSSIZE*ran1(seed))+1;
                                y=(int)((float)SYSSIZE*ran1(seed))+1;
                                z=(int)((float)SYSSIZE*ran1(seed))+1;
                                /* See if the sphere will fit at x,y,z */
                                /* Include dispersion distance when checking */
                             /* to insure requested separation between spheres */
                              count=chksph(x,y,z,radius+dispdist,1,npart+CEM,0);
                                if(tries>MAXTRIES){
  printf("Could not place sphere %d after %d random attempts \n",npart,MAXTRIES);
                                      exit(1);
                                }
                          } while(count!=0);

                          /* place the sphere at x,y,z */
                          npart+=1;
                          if(npart>=NPARTC){
                                printf("Too many spheres being generated \n");
          printf("User needs to increase value of NPARTC at top of C-code\n");
                                exit(1);
                          }
                          /* Allocate space for new particle info */
                     clust[npart]=(struct cluster *)malloc(sizeof(struct cluster));
                          clust[npart]->partid=npart;
                          clust[npart]->clustid=npart;
                          /* Default to cement placement */
                          clust[npart]->partphase=CEMID;
                          clust[npart]->x=x;
                          clust[npart]->y=y;
                          clust[npart]->z=z;
                          clust[npart]->r=radius;
                          clusleft+=1;
                          testgyp=ran1(seed);
                     /* Do not use dispersion distance when placing particle */
                          if(testgyp>probgyp){
                                count=chksph(x,y,z,radius,2,npart+CEM-1,CEMID);
                          }
                          else{
                                /* Place particle as gypsum */
                                count=chksph(x,y,z,radius,2,npart+CEM-1,GYPID);
                                /* Correct phase ID of particle */
                                clust[npart]->partphase=GYPID;
                          }
                          clust[npart]->nextpart=NULL;
                    }
            }
      }
```

```
/* routine to obtain user input and create a starting microstructure */
void create()
/* Calls: gsphere */
/* Called by: main program */
{
	int numsize,sphrad [NUMSIZES];
	long int sphnum [NUMSIZES],inval1;
	int isph,inval;

	do{
printf("Enter number of different size spheres to use(max. is %d) \n",NUMSIZES);
		scanf("%d",&numsize);
		printf("%d \n",numsize);
	}while ((numsize>NUMSIZES)||(numsize<0));
	do{
		printf("Enter dispersion factor for spheres (0-2) \n");
		scanf("%d",&dispdist);
		printf("%d \n",dispdist);
	}while ((dispdist<0)||(dispdist>2));
	do{
		printf("Enter probability for gypsum particles (0.0-1.0) \n");
		scanf("%f",&probgyp);
		printf("%f \n",probgyp);
	} while ((probgyp<0.0)||(probgyp>1.0));

	if((numsize>0)&&(numsize<(NUMSIZES+1))){
       printf("Enter number and radius for each class (largest radius 1st) \n");

	/* Obtain input for each size class of spheres */
		for(isph=0;isph<numsize;isph++){
			printf("Enter number of spheres of class %d \n",isph+1);
			scanf("%ld",&inval1);
			printf("%ld \n",inval1);
			sphnum[isph]=inval1;
			do{
			printf("Enter radius of spheres of class %d \n",isph+1);
				printf("(Integer <=%d please) \n",SYSSIZE/5);
				scanf("%d",&inval);
				printf("%d \n",inval);
			} while ((inval<1)||(inval>(SYSSIZE/5)));
			sphrad[isph]=inval;

		}
		gsphere(numsize,sphnum,sphrad);
	}
}
```

```
/* Routine to draw a particle during flocculation routine */
/* See routine chksph for definition of parameters */
void drawfloc(xin,yin,zin,radd,phasein,phase2)
        int xin,yin,zin,radd,phasein,phase2;
/* Calls: no other routines */
/* Called by: makefloc */
{
        int xp,yp,zp,i,j,k;
        float dist,xdist,ydist,zdist,ftmp;

/* Check all pixels within the digitized sphere volume */
        for(i=xin-radd;(i<=xin+radd);i++){
                xp=i;
                /* use periodic boundary conditions for sphere placement */
                if(xp<1) {xp+=SYSSIZE;}
                else if(xp>SYSSIZE) {xp-=SYSSIZE;}
                ftmp=(float)(i-xin);
                xdist=ftmp*ftmp;
        for(j=yin-radd;(j<=yin+radd);j++){
                yp=j;
                /* use periodic boundary conditions for sphere placement */
                if(yp<1) {yp+=SYSSIZE;}
                else if(yp>SYSSIZE) {yp-=SYSSIZE;}
                ftmp=(float)(j-yin);
                ydist=ftmp*ftmp;
        for(k=zin-radd;(k<=zin+radd);k++){
                zp=k;
                /* use periodic boundary conditions for sphere placement */
                if(zp<1) {zp+=SYSSIZE;}
                else if(zp>SYSSIZE) {zp-=SYSSIZE;}
                ftmp=(float)(k-zin);
                zdist=ftmp*ftmp;

        /* Compute distance from center of sphere to this pixel */
                dist=sqrt(xdist+ydist+zdist);
                if((dist-0.5)<=(float)radd){
                        /* Update both cement and cemreal images */
                        cement [xp] [yp] [zp]=phasein;
                        cemreal [xp] [yp] [zp]=phase2;
                }
        }
        }
        }
}

/* Routine to check particle placement during flocculation */
/* for particle of size radd centered at (xin,yin,zin) */
```

```
/* Returns flag indicating if placement is possible */
int chkfloc(xin,yin,zin,radd)
	int xin,yin,zin,radd;
/* Calls: no other routines */
/* Called by: makefloc */
{
	int nofits,xp,yp,zp,i,j,k;
	float dist,xdist,ydist,zdist,ftmp;

	nofits=0;	/* Flag indicating if placement is possible */

/* Check all pixels within the digitized sphere volume */
	for(i=xin-radd;((i<=xin+radd)&&(nofits==0));i++){
		xp=i;
		/* use periodic boundary conditions for sphere placement */
		if(xp<1) {xp+=SYSSIZE;}
		else if(xp>SYSSIZE) {xp-=SYSSIZE;}
		ftmp=(float)(i-xin);
		xdist=ftmp*ftmp;
	for(j=yin-radd;((j<=yin+radd)&&(nofits==0));j++){
		yp=j;
		/* use periodic boundary conditions for sphere placement */
		if(yp<1) {yp+=SYSSIZE;}
		else if(yp>SYSSIZE) {yp-=SYSSIZE;}
		ftmp=(float)(j-yin);
		ydist=ftmp*ftmp;
	for(k=zin-radd;((k<=zin+radd)&&(nofits==0));k++){
		zp=k;
		/* use periodic boundary conditions for sphere placement */
		if(zp<1) {zp+=SYSSIZE;}
		else if(zp>SYSSIZE) {zp-=SYSSIZE;}
		ftmp=(float)(k-zin);
		zdist=ftmp*ftmp;

	/* Compute distance from center of sphere to this pixel */
		dist=sqrt(xdist+ydist+zdist);
		if((dist-0.5)<=(float)radd){
			if((cement [xp] [yp] [zp] !=POROSITY)){
				/* Record ID of particle hit */
				nofits=cement [xp] [yp] [zp];
			}
		}
		/* Check for overlap with aggregate */
		if((abs(xp-((float)(SYSSIZE+1)/2.0)))<((float)aggsize/2.0)){
			nofits=AGG;
		}
	}
```

```
        }
        }

        /* return flag indicating if sphere will fit */
        return(nofits);
}

/* routine to perform flocculation of particles */
void makefloc()
/* Calls: drawfloc, chkfloc, ran1 */
/* Called by: main program */
{
        int partdo,numfloc;
        int nstart;
        int nleft,ckall;
        int xm,ym,zm,moveran;
        int xp,yp,zp,rp,clushit,valkeep;
        int iclus,cluspart[NPARTC];
        struct cluster *parttmp,*partpoint,*partkeep;

        nstart=npart; /* Counter of number of flocs remaining */
        for(iclus=1;iclus<=npart;iclus++){
                cluspart[iclus]=iclus;
        }
        do{
          printf("Enter number of flocs desired at end of routine (>0) \n");
                scanf("%d",&numfloc);
                printf("%d\n",numfloc);
        } while (numfloc<=0);

        while(nstart>numfloc){
                nleft=0;

                /* Try to move each cluster in turn */
                for(iclus=1;iclus<=npart;iclus++){
                        if(clust[iclus]==NULL){
                                nleft+=1;
                        }
                        else{
                                xm=ym=zm=0;
                /* Generate a random move in one of 6 principal directions */
                moveran=6.*ran1(seed);
                switch(moveran){
                                case 0:
                                        xm=1;
                                        break;
                                case 1:
```

```
                                xm=(-1);
                                break;
                        case 2:
                                ym=1;
                                break;
                        case 3:
                                ym=(-1);
                                break;
                        case 4:
                                zm=1;
                                break;
                        case 5:
                                zm=(-1);
                                break;
                        default:
                                break;
        }

        /* First erase all particles in cluster */
        partpoint=clust[iclus];
        while(partpoint!=NULL){
                xp=partpoint->x;
                yp=partpoint->y;
                zp=partpoint->z;
                rp=partpoint->r;
                drawfloc(xp,yp,zp,rp,0,0);
                partpoint=partpoint->nextpart;
        }

        ckall=0;
        /* Now try to draw cluster at new location */
        partpoint=clust[iclus];
        while((partpoint!=NULL)&&(ckall==0)){
                xp=partpoint->x+xm;
                yp=partpoint->y+ym;
                zp=partpoint->z+zm;
                rp=partpoint->r;
                ckall=chkfloc(xp,yp,zp,rp);
                partpoint=partpoint->nextpart;
        }

        if(ckall==0){
        /* Place cluster particles at new location */
                partpoint=clust[iclus];
                while(partpoint!=NULL){
                        xp=partpoint->x+xm;
                        yp=partpoint->y+ym;
```

```
                        zp=partpoint->z+zm;
                        rp=partpoint->r;
                        valkeep=partpoint->partphase;
                        partdo=partpoint->partid;
                        drawfloc(xp,yp,zp,rp,partdo+CEM-1,valkeep);
                        /* Update particle location */
                        partpoint->x=xp;
                        partpoint->y=yp;
                        partpoint->z=zp;
                        partpoint=partpoint->nextpart;
                }
        }
        else{
                /* A cluster or aggregate was hit */
                /* Draw particles at old location */
                partpoint=clust[iclus];

                /* partkeep stores pointer to last particle in list */
                while(partpoint!=NULL){
                        xp=partpoint->x;
                        yp=partpoint->y;
                        zp=partpoint->z;
                        rp=partpoint->r;
                        valkeep=partpoint->partphase;
                        partdo=partpoint->partid;
                        drawfloc(xp,yp,zp,rp,partdo+CEM-1,valkeep);
                        partkeep=partpoint;
                        partpoint=partpoint->nextpart;
                }
                /* Determine the cluster hit */
                if(ckall!=AGG){
                        clushit=cuspart[ckall-CEM+1];
/* Move all of the particles from cluster clushit to cluster iclus */
                        parttmp=clust[clushit];
                        /* Attach new cluster to old one */
                        partkeep->nextpart=parttmp;
                        while(parttmp!=NULL){
                                cluspart[parttmp->partid]=iclus;
                  /* Relabel all particles added to this cluster */
                                parttmp->clustid=iclus;
                                parttmp=parttmp->nextpart;
                        }
                        /* Disengage the cluster that was hit */
                        clust[clushit]=NULL;
                        nstart-=1;
                }
        }
        }
```

```
        }
   printf("Number left was %d but number of clusters is %d \n",nleft,nstart);
        }       /* end of while loop */
        clusleft=nleft;
}

/* routine to assess global phase fractions present in 3-D system */
void measure()
/* Calls: no other routines */
/* Called by: main program */
{
        long int npor,ngyp,ncem,nagg;
        int i,j,k,valph;

/* counters for the various phase fractions */
        npor=0;
        ngyp=0;
        ncem=0;
        nagg=0;

/* Check all pixels in 3-D microstructure */
        for(i=1;i<=SYSSIZE;i++){
        for(j=1;j<=SYSSIZE;j++){
        for(k=1;k<=SYSSIZE;k++){
                valph=cemreal [i] [j] [k];
                if(valph==POROSITY) {npor+=1;}
                else if(valph==CEMID){ncem+=1;}
                else if(valph==GYPID){ngyp+=1;}
                else if(valph==AGG) {nagg+=1;}
        }
        }
        }

/* Output results */
        printf("Porosity= %ld \n",npor);
        printf("Cement= %ld \n",ncem);
        printf("Gypsum= %ld \n",ngyp);
        printf("Aggregate= %ld \n",nagg);
}

/* Routine to measure phase fractions as a function of distance from */
/* aggregate surface      */
void measagg()
/* Calls: no other routines */
/* Called by: main program */
{
        int phase [8],ptot;
```

```
        int icnt,ixlo,ixhi,iy,iz,phid,idist;

        printf("Distance  Porosity  Cement  Gypsum  \n");

 /* Increase distance from aggregate in increments of one */
        for(idist=1;idist<=(SYSSIZE-aggsize)/2;idist++){
                /* Pixel left of aggregate surface */
                ixlo=((SYSSIZE-aggsize+2)/2)-idist;
                /* Pixel right of aggregate surface */
                ixhi=((SYSSIZE+aggsize)/2)+idist;

        /* Initialize phase counts for this distance */
                for(icnt=0;icnt<8;icnt++){
                        phase[icnt]=0;
                }
                ptot=0;

/* Check all pixels which are this distance from aggregate surface */
                for(iy=1;iy<=SYSSIZE;iy++){
                for(iz=1;iz<=SYSSIZE;iz++){
                        phid=cemreal [ixlo] [iy] [iz];
                        ptot+=1;
                        if(phid<8){
                                phase[phid]+=1;
                        }
                        phid=cemreal [ixhi] [iy] [iz];
                        ptot+=1;
                        if(phid<8){
                                phase[phid]+=1;
                        }
                }
                }

                /* Output results for this distance from surface */
    printf("%d   %d   %d  %d\n",idist,phase[0],phase[CEMID],phase[GYPID]);

        }
}

/* routine to assess the connectivity (percolation) of a single phase */
/* Two matrices are used here: one for the current burnt locations */
/*                 the other to store the newly found burnt locations */
void connect()
/* Calls: no other routines */
/* Called by: main program */
{
        long int inew,ntop,nthrough,ncur,nnew,ntot;
```

```
   int i,j,k,nmatx[29000],nmaty[29000],nmatz[29000];
int xcn,ycn,zcn,npix,x1,y1,z1,igood,nnewx[29000],nnewy[29000],nnewz[29000];
   int jnew,icur;

   do{
           printf("Enter phase to analyze 0) pores 1) Cement  \n");
           scanf("%d",&npix);
           printf("%d \n",npix);
   } while ((npix!=0)&&(npix!=1));

   /* counters for number of pixels of phase accessible from top surface */
   /* and number which are part of a percolated pathway */
   ntop=0;
   nthrough=0;

   /* percolation is assessed from top to bottom only */
   /* and burning algorithm is nonperiodic in x and y directions */
   k=1;
   for(i=1;i<=SYSSIZE;i++){
           for(j=1;j<=SYSSIZE;j++){
                   ncur=0;
                   ntot=0;
                   igood=0;     /* Indicates if bottom has been reached */
if(((cement [i] [j] [k]==npix)&&((cement [i] [j] [SYSSIZE]==npix)||
(cement [i] [j] [SYSSIZE]==(npix+BURNT))))||((cement [i] [j] [SYSSIZE]>=CEM)&&
(cement [i] [j] [k]>=CEM)&&(cement [i] [j] [k]<BURNT)&&(npix==1))){
                           /* Start a burn front */
                           cement [i] [j] [k]+=BURNT;
                           ntot+=1;
                           ncur+=1;
                           /* burn front is stored in matrices nmat* */
                           /* and nnew* */
                           nmatx[ncur]=i;
                           nmaty[ncur]=j;
                           nmatz[ncur]=1;
                         /* Burn as long as new (fuel) pixels are found */
                           do{
                                   nnew=0;
                                   for(inew=1;inew<=ncur;inew++){
                                           xcn=nmatx[inew];
                                           ycn=nmaty[inew];
                                           zcn=nmatz[inew];

                                           /* Check all six neighbors */
                                           for(jnew=1;jnew<=6;jnew++){
                                                   x1=xcn;
                                                   y1=ycn;
```

```
z1=zcn;
if(jnew==1){
        x1-=1;
        if(x1<1){
            x1+=SYSSIZE;
        }
}
else if(jnew==2){
        x1+=1;
        if(x1>SYSSIZE){
            x1-=SYSSIZE;
        }
}
else if(jnew==3){
        y1-=1;
        if(y1<1){
            y1+=SYSSIZE;
        }
}
else if(jnew==4){
        y1+=1;
        if(y1>SYSSIZE){
            y1-=SYSSIZE;
        }
}
else if(jnew==5){
        z1-=1;
}
else if(jnew==6){
        z1+=1;
}

/* Nonperiodic in z direction so be sure to remain in the 3-D box */
if((z1>=1)&&(z1<=SYSSIZE)){
if((cement [x1] [y1] [z1]==npix)||((cement [x1] [y1] [z1]>=CEM)&&
(cement [x1] [y1] [z1]<BURNT)&&(npix==1))){
        ntot+=1;
    cement [x1] [y1] [z1]+=BURNT;
        nnew+=1;
    if(nnew>=29000){
printf("error in size of nnew \n");
    }
        nnewx[nnew]=x1;
        nnewy[nnew]=y1;
        nnewz[nnew]=z1;
/* See if bottom of system has been reached */
    if(z1==SYSSIZE){
```

```
                                                                        igood=1;
                                                                  }
                                                                  }
                                                        }
                                                        }
                                                }
                                                if(nnew>0){
                                                        ncur=nnew;
                                                /* update the burn front matrices */
                                                        for(icur=1;icur<=ncur;icur++){
                                                                nmatx[icur]=nnewx[icur];
                                                                nmaty[icur]=nnewy[icur];
                                                                nmatz[icur]=nnewz[icur];
                                                        }
                                                }
                                        }while (nnew>0);

                                        ntop+=ntot;
                                        if(igood==1){
                                                nthrough+=ntot;
                                        }
                                }
                        }
                }

        printf("Phase ID= %d \n",npix);
        printf("Number accessible from top= %ld \n",ntop);
        printf("Number contained in through pathways= %ld \n",nthrough);

        /* return the burnt sites to their original phase values */
        for(i=1;i<=SYSSIZE;i++){
        for(j=1;j<=SYSSIZE;j++){
        for(k=1;k<=SYSSIZE;k++){
                if(cement [i] [j] [k]>=BURNT){
                        cement [i] [j] [k]-=BURNT;
                }
        }
        }
        }
}

/* Routine to output final microstructure to file */
void outmic()
/* Calls: no other routines */
/* Called by: main program */
{
        FILE *outfile,*partfile;
```

```
        char filen[80],filepart[80];
        int ix,iy,iz,valout;

        printf("Enter name of file to save microstructure to \n");
        scanf("%s",filen);
        printf("%s\n",filen);

        outfile=fopen(filen,"w");

        printf("Enter name of file to save particle IDs to \n");
        scanf("%s",filepart);
        printf("%s\n",filepart);

        partfile=fopen(filepart,"w");

        for(iz=1;iz<=SYSSIZE;iz++){
        for(iy=1;iy<=SYSSIZE;iy++){
        for(ix=1;ix<=SYSSIZE;ix++){
                valout=cemreal[ix][iy][iz];
                fprintf(outfile,"%1d\n",valout);
                valout=cement[ix][iy][iz];
                if(valout<0){valout=0;}
                fprintf(partfile,"%d\n",valout);
        }
        }
        }
        fclose(outfile);
        fclose(partfile);
}

main(){
        int userc;      /* User choice from menu */
        int nseed,ig,jg,kg;

        printf("Enter random number seed value \n");
        scanf("%d",&nseed);
        printf("%d \n",nseed);
        seed=(&nseed);

        /* Initialize counters and system parameters */
        npart=0;
        aggsize=0;
        clusleft=0;

        /* clear the 3-D system to all porosity to start */
        for(ig=1;ig<=SYSSIZE;ig++){
        for(jg=1;jg<=SYSSIZE;jg++){
```

```
        for(kg=1;kg<=SYSSIZE;kg++){
                cement [ig] [jg] [kg]=POROSITY;
                cemreal [ig] [jg] [kg]=POROSITY;
        }
        }
        }

        /* present menu and execute user choice */
        do{
printf(" \n Input User Choice \n");
printf("1) Exit \n");
printf("2) Add spherical particles (cement and gypsum) to microstructure \n");
printf("3) Flocculate system by reducing number of particle clusters \n");
printf("4) Measure phase fractions \n");
printf("5) Add an aggregate to the microstructure \n");
printf("6) Measure single phase connectivity (pores or solids) \n");
printf("7) Measure phase fractions vs. distance from aggregate \n");
printf("8) Output microstructure to file \n");

                scanf("%d",&userc);
                printf("%d \n",userc);
                fflush(stdout);

                switch (userc) {
                        case 2:
                                create();
                                break;
                        case 3:
                                makefloc();
                                break;
                        case 4:
                                measure();
                                break;
                        case 5:
                                addagg();
                                break;
                        case 6:
                                connect();
                                break;
                        case 7:
                                if(aggsize!=0){
                                measagg();
        }
        else{
        printf("No aggregate present. \n");
        }
                                break;
```

```
                    case 8:
                            outmic();
                            break;
                    default:
                            break;
            }
        } while (userc!=1);
}
```

B.2 Listing for rand3d.f

```
C
C     Program RAND3D.F: to create a 100*100*100
C     microstructure based on filtering Gaussian
C     noise with autocorrelation filter of a 2-D
C     image
C     Programmer: Dale Bentz (dale.bentz@nist.gov)
C     Date: 1993
C     1995: Modified to filter random particle images
C     for cement hydration model
C
      INTEGER SEED,SYSSIZE,IRES
      REAL SNEW,S2,SS,SDIFF,PHASEIN,PHASEOUT,XTMP,YTMP
      REAL NORMM(100,100,100),RES(100,100,100)
      REAL MASK(100,100,100)
      REAL FILTER(31,31,31)
      INTEGER R(60)
      REAL S(60),XR(60),SUM(501)
      REAL T1,T2,X1,X2,U1,U2,XRAD,RESMAX,RESMIN
      INTEGER R1,R2,I1,I2,I3,I,J,K,J1,K1
      INTEGER IDO,III,JJJ,IX,IY,IZ,INDEX
      REAL PI,XTOT,FILVAL,RADIUS,XPT,SECT,SUMTOT,VCRIT
      CHARACTER*80 FILEN,FILEM

      PI=3.1415926
      SYSSIZE=100
      WRITE(6,*)'Input random seed'
      READ(5,*)SEED
      WRITE(6,*)SEED
C
C     In this version, only a portion of the original 3-D image is filtered
C     so that for instance, silicates may be separated into C3S and C2S
C     without modification of the aluminate and gypsum phases
C
      WRITE(6,*)'Input existing phase assignment for matching'
      READ(5,*)PHASEIN
```

```
      WRITE(6,*)PHASEIN
      WRITE(6,*)'Input phase assignment to be created'
      READ(5,*)PHASEOUT
      WRITE(6,*)PHASEOUT
C
C      Read in mask image (particles) from file
C
      WRITE(6,*)'Enter name of cement microstructure image file'
      READ(5,*)FILEM
      OPEN(13,FILE=FILEM,STATUS='OLD')

      DO 19 I=1,SYSSIZE
      DO 19 J=1,SYSSIZE
      DO 19 K=1,SYSSIZE
   19 READ(13,*)MASK(I,J,K)

      CLOSE(13)
C
C      Create the gaussian noise image
C      by appropriately modifying uniform noise
C      image
C      Source: Law, A.M, and Kelton, W.D.,
C      Simulation Modeling and Analysis, McGraw-Hill, 1982.
C
      I1=1
      I2=1
      I3=1
      DO 70 I=1,500000
      U1=RAN1(SEED)
      U2=RAN1(SEED)
      T1=2.*PI*U2
      T2=(-2.*LOG(U1))**0.5
      X1=COS(T1)*(T2)
      X2=SIN(T1)*(T2)
      NORMM(I1,I2,I3)=X1
      I1=I1+1
      IF(I1.GT.SYSSIZE) THEN
        I1=1
        I2=I2+1
        IF(I2.GT.SYSSIZE) THEN
          I2=1
          I3=I3+1
        ENDIF
      ENDIF
      NORMM(I1,I2,I3)=X2
      I1=I1+1
      IF(I1.GT.SYSSIZE) THEN
```

```
          I1=1
          I2=I2+1
          IF(I2.GT.SYSSIZE) THEN
            I2=1
            I3=I3+1
          ENDIF
        ENDIF
   70   CONTINUE
        CLOSE(14)
C
C        Now convolve with the autocorrelation function of the goal 2-D image
C
        WRITE(6,*)'Enter filename to read in autocorrelation from'
        READ(5,*)FILEN
        OPEN(8,FILE=FILEN,STATUS='OLD')
        READ(8,*)IDO
        WRITE(6,*)IDO
        DO 95 I=1,IDO
        READ(8,*)R(I),S(I)
        XR(I)=R(I)
   95   CONTINUE
        CLOSE(8)
        SS=S(1)
        S2=SS*SS
C
C       Load up the convolution matrix (31*31*31 in size)
C
        OPEN(10,FILE='FILTER3D.IMG')
        SDIFF=SS-S2
        DO 14 I=0,30
        III=I*I
        DO 13 J=0,30
        JJJ=J*J
        DO 12 K=0,30
        XTMP=III+JJJ+K*K
C
C       Use Linear interpolation to estimate S(x,y,z) from available S(r)
C
        RADIUS=SQRT(XTMP)
        R1=RADIUS+1
        R2=R1+1
        XRAD=RADIUS+1-R1
        FILVAL=S(R1)+(S(R2)-S(R1))*XRAD
        FILTER(I+1,J+1,K+1)=(FILVAL-S2)/(SDIFF)
   12   WRITE(10,*)FILTER(I+1,J+1,K+1)
   13   CONTINUE
   14   CONTINUE
```

```
      CLOSE(10)
C
C     Now filter the image maintaining periodic boundaries
C
C     Store minimum (RESMIN) and maximum (RESMAX) values for later binning
C
      RESMAX=0.0
      RESMIN=1.0
      DO 97 I=1,SYSSIZE
      DO 97 J=1,SYSSIZE
      DO 97 K=1,SYSSIZE
      RES(I,J,K)=0.0
      DO 98 IX=1,31
      I1=I+IX-1
C
C     Periodic boundaries
C
      IF(I1.LT.1) THEN
        I1=I1+SYSSIZE
      ELSE IF(I1.GT.SYSSIZE) THEN
        I1=I1-SYSSIZE
      ENDIF
      DO 98 IY=1,31
      J1=J+IY-1
      IF(J1.LT.1) THEN
        J1=J1+SYSSIZE
      ELSE IF(J1.GT.SYSSIZE) THEN
        J1=J1-SYSSIZE
      ENDIF
      DO 98 IZ=1,31
      K1=K+IZ-1
      IF(K1.LT.1) THEN
        K1=K1+SYSSIZE
      ELSE IF(K1.GT.SYSSIZE) THEN
        K1=K1-SYSSIZE
      ENDIF
      RES(I,J,K)=RES(I,J,K)+NORMM(I1,J1,K1)*FILTER(IX,IY,IZ)
  98  CONTINUE
C
C     Check for min/max if pixel is part of proper phase
C
      IF(MASK(I,J,K).EQ.PHASEIN) THEN
        IF(RES(I,J,K).GT.RESMAX) THEN
          RESMAX=RES(I,J,K)
        ENDIF
        IF(RES(I,J,K).LT.RESMIN) THEN
          RESMIN=RES(I,J,K)
```

```
        ENDIF
      ENDIF
   97 CONTINUE
C
C      Now threshold the image
C
      WRITE(6,*)'Input desired threshold phase fraction'
      READ(5,*)XPT
      WRITE(6,*)XPT
C
C     Bin the resultant (filtered) values into 500 bins
C     so that adequate binarization (thresholding) can be performed
C
      SECT=(RESMAX-RESMIN)/500.0
      WRITE(6,*)'SECT is',SECT
      WRITE(6,*)'RESMAX and RESMIN are ',RESMAX,RESMIN
      DO 34 I=1,500
   34 SUM(I)=0.0
C
C     Fill in the 500-bin histogram
C
      XTOT=0.0
      DO 33 I=1,SYSSIZE
      DO 33 J=1,SYSSIZE
      DO 33 K=1,SYSSIZE
      IF(MASK(I,J,K).EQ.PHASEIN) THEN
        XTOT=XTOT+1.0
        INDEX=1+(RES(I,J,K)-RESMIN)/SECT
        IF(INDEX.GT.500) THEN
          INDEX=500
        ENDIF
        SUM(INDEX)=SUM(INDEX)+1.0
      ENDIF
   33 CONTINUE
C
C     Determine which bin to choose for correct thresholding
C
      SUMTOT=0.0
      DO 35 I=1,500
      SUMTOT=SUMTOT+SUM(I)/(XTOT)
      IF(SUMTOT.GT.XPT) THEN
        YTMP=I
        VCRIT=RESMIN+(RESMAX-RESMIN)*(YTMP-0.5)/500.
        GOTO 36
      ENDIF
   35 CONTINUE
```

```
 36  WRITE(6,*)'VCRIT is',VCRIT
     IRES=0
C
C    Perform the segmentation and output the resultant image
C
     OPEN(9,FILE='IMAGE3D.OUT')
     DO 32 I=1,SYSSIZE
     DO 32 J=1,SYSSIZE
     DO 32 K=1,SYSSIZE
C
C    Only set values that were originally the phase of choice
C
     IF(MASK(I,J,K).EQ.PHASEIN) THEN
       IF(RES(I,J,K).GT.VCRIT) THEN
         RES(I,J,K)=PHASEOUT
       ELSE
         RES(I,J,K)=PHASEIN
         IRES=IRES+1
       ENDIF
     ELSE
       RES(I,J,K)=MASK(I,J,K)
     ENDIF
     WRITE(9,*)INT(RES(I,J,K))
 32  CONTINUE
     CLOSE(9)
     WRITE(6,*)'Solids are ',FLOAT(IRES)/1000000.
     STOP
     END
     FUNCTION RAN1(IDUM)
C
C    Portable random number generator, RAN1
C    To generate uniform random deviates between 0 and 1.0
C    From: Computers in Physics, Vol. 6, (5), 1992, 522-524
C          Press and Teukolsky
C
C    CAll with idum set to a negative number to initialize
C    RNMAX should approximate the largest floating value that
C    is less than 1.0
     INTEGER IDUM,IA,IM,IQ,IR,NTAB,NDIV
     REAL RAN1,AM,EPS,RNMAX
     PARAMETER (IA=16807,IM=2147483647,AM=1./IM,IQ=127773,IR=2836,
    +     NTAB=32,NDIV=1+(IM-1)/NTAB,EPS=1.2E-7,RNMAX=1.-EPS)

     INTEGER J,K,IV(NTAB),IY
     SAVE IV,IY
     DATA IV /NTAB*0/, IY /0/
```

```
      IF(IDUM.LE.0.OR.IY.EQ.0) THEN
        IF(IDUM.LE.0) THEN
          IDUM=MAX(-IDUM,1)
        ENDIF
        DO 11 J=NTAB+8,1,-1
          K=IDUM/IQ
          IDUM=IA*(IDUM-K*IQ)-IR*K
          IF(IDUM.LT.0) IDUM=IDUM+IM
          IF(J.LE.NTAB) IV(J)=IDUM
11      CONTINUE
        IY=IV(1)
      ENDIF
      K=IDUM/IQ
      IDUM=IA*(IDUM-K*IQ)-IR*K
      IF(IDUM.LT.0) IDUM=IDUM+IM
      J=1+IY/NDIV
      IY=IV(J)
      IV(J)=IDUM
      RAN1=MIN(AM*IY,RNMAX)
      RETURN
      END
```

B.3 Listing for stat3d.c

```
/************************************************************************/
/*                                                                      */
/*      Program: stat3d.c                                               */
/*      Purpose: To read in a 3-D image and output phase volumes        */
/*               and report the volume and pore-exposed surface area    */
/*               fractions                                              */
/*               Assumes that input image contains only the values 0-6  */
/*      Programmer: Dale P. Bentz                                       */
/*                  NIST                                                */
/*                  Building 226 Room B-350                             */
/*                  Gaithersburg, MD  20899-0001                        */
/*                  Phone: (301) 975-5865                               */
/*                  E-mail: dale.bentz@.nist.gov                        */
/*                                                                      */
/************************************************************************/
#include <stdio.h>
#include <math.h>

#define ISIZE 100


main(){
```

```
        static int mic [ISIZE] [ISIZE] [ISIZE];
        int valin,ix,iy,iz;
        int ix1,iy1,iz1,k;
        long int surftot,volume[7],surface [7];
        FILE *infile;
        char filen[80];

        printf("Enter name of file to open \n");
        scanf("%s",filen);
        printf("%s \n",filen);

        for(ix=0;ix<=6;ix++){
                volume[ix]=surface[ix]=0;
        }

        infile=fopen(filen,"r");

/* Read in image and accumulate volume totals */
        for(iz=0;iz<ISIZE;iz++){
        for(iy=0;iy<ISIZE;iy++){
        for(ix=0;ix<ISIZE;ix++){
                fscanf(infile,"%d",&valin);
                mic [ix] [iy] [iz]=valin;
                volume[valin]+=1;
        }
        }
        }

        fclose(infile);

        for(iz=0;iz<ISIZE;iz++){
        for(iy=0;iy<ISIZE;iy++){
        for(ix=0;ix<ISIZE;ix++){
                if(mic [ix] [iy] [iz]!=0){
                        valin=mic [ix] [iy] [iz];
                /* Check six neighboring pixels for porosity */
                        for(k=1;k<=6;k++){

                                switch (k){
                                        case 1:
                                                ix1=ix-1;
                                                if(ix1<0){ix1+=ISIZE;}
                                                iy1=iy;
                                                iz1=iz;
                                                break;
                                        case 2:
                                                ix1=ix+1;
```

```
                                        if(ix1>=ISIZE){ix1-=ISIZE;}
                                        iy1=iy;
                                        iz1=iz;
                                        break;
                                case 3:
                                        iy1=iy-1;
                                        if(iy1<0){iy1+=ISIZE;}
                                        ix1=ix;
                                        iz1=iz;
                                        break;
                                case 4:
                                        iy1=iy+1;
                                        if(iy1>=ISIZE){iy1-=ISIZE;}
                                        ix1=ix;
                                        iz1=iz;
                                        break;
                                case 5:
                                        iz1=iz-1;
                                        if(iz1<0){iz1+=ISIZE;}
                                        iy1=iy;
                                        ix1=ix;
                                        break;
                                case 6:
                                        iz1=iz+1;
                                        if(iz1>=ISIZE){iz1-=ISIZE;}
                                        iy1=iy;
                                        ix1=ix;
                                        break;
                                default:
                                        break;
                        }
        if((ix1<0)||(iy1<0)||(iz1<0)||(ix1>=ISIZE)||(iy1>=ISIZE)||(iz1>=ISIZE)){
                                printf("%d %d %d \n",ix1,iy1,iz1);
                                exit(1);
                        }
                        if(mic[ix1] [iy1] [iz1]==0){
                                surface[valin]+=1;
                        }
                }
        }
        }
        }
        }

        printf("Phase    Volume      Surface     Surface Fraction \n");
/* Only include non-gypsum phases in surface area fraction calculation */
        surftot=surface[1]+surface[2]+surface[3]+surface[4];
```

```
printf("Total surface area count (not including gypsum and aggregate) is %ld\n",
            surftot);
        for(k=0;k<=6;k++){
                if(k<=4){
        printf("%d      %8ld     %8ld       %.5f\n",k,volume[k],surface[k],
                (float)surface[k]/(float)surftot);
                }
                else{
                        printf("%d      %8ld     %8ld\n",k,volume[k],surface[k]);
                }
        }
}
```

B.4 Listing for sinter3d.c

```
/******************************************************************/
/*                                                                */
/*      Program sinter3d.c                                        */
/*      To simulate 3-D curvature controlled sintering            */
/*      of a loosely packed powder                                */
/*      This version to sinter between two phases in              */
/*      a multi-phase system                                      */
/*      This version is periodic in all three directions          */
/*      Application: Modify local structure of generated          */
/*      3-D porous media by matching a specific hydraulic         */
/*      radius                                                    */
/*                                                                */
/*      Programmer: Dale P. Bentz                                 */
/*                  NIST                                          */
/*                  Building 226 Room B-350                       */
/*                  Gaithersburg, MD  20899-0001                  */
/*                  Phone: (301) 975-5865                         */
/*                  E-mail: dale.bentz@.nist.gov                  */
/*      Date: Summer 1994                                         */
/*                                                                */
/******************************************************************/

#include <stdio.h>
#include <math.h>

#define SYSSIZE 100
#define MAXCYC 1000    /* maximum sintering cycles to use */
#define MAXSPH 10000   /* maximum number of elements in a spherical template */

/* array phase stores the microstructure representation */
/* array curvature stores the local curvature values */
```

```
static unsigned short int phase [SYSSIZE+1] [SYSSIZE+1] [SYSSIZE+1];
static unsigned short int curvature [SYSSIZE+1] [SYSSIZE+1] [SYSSIZE+1];
int nsph,xsph[MAXSPH],ysph[MAXSPH],zsph[MAXSPH];
int *seed;
long int nsolid[500],nair[500];

/* Random number generator ran1 from Computers in Physics */
/* Volume 6 No. 5, 1992, 522-524, Press and Teukolsky */
/* To generate real random numbers 0.0-1.0 */
/* Should be seeded with a negative integer */
#define IA 16807
#define IM 2147483647
#define IQ 127773
#define IR 2836
#define NTAB 32
#define EPS (1.2E-07)
#define MAX(a,b) (a>b)?a:b
#define MIN(a,b) (a<b)?a:b

double ran1(idum)
int *idum;
/* Calls: no routines */
/* Called by: gsphere,makefloc */
{
        int j,k;
        static int iv[NTAB],iy=0;
        void nrerror();
        static double NDIV = 1.0/(1.0+(IM-1.0)/NTAB);
        static double RNMX = (1.0-EPS);
        static double AM = (1.0/IM);

        if ((*idum <= 0) || (iy == 0)) {
                *idum = MAX(-*idum,*idum);
                for(j=NTAB+7;j>=0;j--) {
                        k = *idum/IQ;
                        *idum = IA*(*idum-k*IQ)-IR*k;
                        if(*idum < 0) *idum += IM;
                        if(j < NTAB) iv[j] = *idum;
                }
                iy = iv[0];
        }
        k = *idum/IQ;
        *idum = IA*(*idum-k*IQ)-IR*k;
        if(*idum<0) *idum += IM;
        j = iy*NDIV;
        iy = iv[j];
        iv[j] = *idum;
```

```
        return MIN(AM*iy,RNMX);
}
#undef IA
#undef IM
#undef IQ
#undef IR
#undef NTAB
#undef EPS
#undef MAX
#undef MIN

/* routine to create a template for the sphere of interest of radius size */
/* to be used in curvature evaluation */
/* Called by: runsint */
/* Calls no other routines */
int maketemp(size)
        int size;
{
        int icirc,xval,yval,zval;
        float xtmp,ytmp;
        float dist;

/* determine and store the locations of all pixels in the 3-D sphere */
        icirc=0;
        for(xval=(-size);xval<=size;xval++){
                xtmp=(float)(xval*xval);
        for(yval=(-size);yval<=size;yval++){
                ytmp=(float)(yval*yval);
        for(zval=(-size);zval<=size;zval++){
                dist=sqrt(xtmp+ytmp+(float)(zval*zval));
                if(dist<=((float)size+0.5)){
                        icirc+=1;
                        if(icirc>=MAXSPH){
                            printf("Too many elements in sphere \n");
                            printf("Must change value of MAXSPH parameter \n");
                            printf("Currently set at %d \n",MAXSPH);
                            exit(1);
                        }
                        xsph[icirc]=xval;
                        ysph[icirc]=yval;
                        zsph[icirc]=zval;
                }
        }
        }
        }

/* return the number of pixels contained in sphere of radius (size+0.5) */
```

```
        return(icirc);
}

/* routine to count phase fractions (porosity and solids) */
/* Called by main routine */
/* Calls no other routines */
void phcount()
{
        long int npore,nsolid [7];
        int ix,iy,iz;

        npore=0;
        nsolid[1]=nsolid[2]=nsolid[3]=nsolid[4]=nsolid[5]=nsolid[6]=0;
        /* check all pixels in the 3-D system */
        for(ix=0;ix<SYSSIZE;ix++){
        for(iy=0;iy<SYSSIZE;iy++){
        for(iz=0;iz<SYSSIZE;iz++){
                if(phase [ix] [iy] [iz]==0){
                        npore+=1;
                }
                else{
                        nsolid[phase [ix] [iy] [iz]]+=1;
                }
        }
        }
        }

        printf("Pores are: %ld \n",npore);
        printf("Solids are: %ld %ld %ld %ld %ld %ld\n",nsolid[1],nsolid[2],
            nsolid[3],nsolid[4],nsolid[5],nsolid[6]);
}

/* routine to return number of surface faces exposed to porosity */
/* for pixel located at (xin,yin,zin) */
/* Called by rhcalc */
/* Calls no other routines */
int surfpix(xin,yin,zin)
        int xin,yin,zin;
{
        int npix,ix1,iy1,iz1;

        npix=0;

        /* check each of the six immediate neighbors */
        /* using periodic boundary conditions */
        ix1=xin-1;
        if(ix1<0){ix1+=SYSSIZE;}
```

```
	if(phase[ix1][yin][zin]==0){
		npix+=1;
	}
	ix1=xin+1;
	if(ix1>=SYSSIZE){ix1-=SYSSIZE;}
	if(phase[ix1][yin][zin]==0){
		npix+=1;
	}
	iy1=yin-1;
	if(iy1<0){iy1+=SYSSIZE;}
	if(phase[xin][iy1][zin]==0){
		npix+=1;
	}
	iy1=yin+1;
	if(iy1>=SYSSIZE){iy1-=SYSSIZE;}
	if(phase[xin][iy1][zin]==0){
		npix+=1;
	}
	iz1=zin-1;
	if(iz1<0){iz1+=SYSSIZE;}
	if(phase[xin][yin][iz1]==0){
		npix+=1;
	}
	iz1=zin+1;
	if(iz1>=SYSSIZE){iz1-=SYSSIZE;}
	if(phase[xin][yin][iz1]==0){
		npix+=1;
	}
	return(npix);
}


/* routine to return the current hydraulic radius for phase phin */
/* Calls surfpix */
/* Called by runsint */
float rhcalc(phin)
	int phin;
{
	int ix,iy,iz;
	long int porc,surfc;
	float rhval;

	porc=surfc=0;

	/* Check all pixels in the 3-D volume */
	for(ix=0;ix<SYSSIZE;ix++){
	for(iy=0;iy<SYSSIZE;iy++){
	for(iz=0;iz<SYSSIZE;iz++){
```

```
			if(phase [ix] [iy] [iz]==phin){
				porc+=1;
				surfc+=surfpix(ix,iy,iz);
			}
	}
	}
	}

	printf("Pore count is %ld \n",porc);
	printf("Surface count is %ld \n",surfc);
	rhval=(float)porc*6./(4.*(float)surfc);
	printf("Hydraulic radius is %f \n",rhval);
	return(rhval);
}

/* routine to return count of pixels in a spherical template which are phase */
/* phin or porosity (phase=0) */
/* Calls no other routines */
/* Called by sysinit */
int countem(xp,yp,zp,phin)
	int xp,yp,zp,phin;
{
	int xc,yc,zc;
	int cumnum,ic;

	cumnum=0;
	for(ic=1;ic<=nsph;ic++){
		xc=xp+xsph[ic];
		yc=yp+ysph[ic];
		zc=zp+zsph[ic];
		/* Use periodic boundaries */
		if(xc<0){xc+=SYSSIZE;}
		else if(xc>=SYSSIZE){xc-=SYSSIZE;}
		if(yc<0){yc+=SYSSIZE;}
		else if(yc>=SYSSIZE){yc-=SYSSIZE;}
		if(zc<0){zc+=SYSSIZE;}
		else if(zc>=SYSSIZE){zc-=SYSSIZE;}

		if((xc!=xp)||(yc!=yp)||(zc!=zp)){

		if((phase [xc] [yc] [zc]==phin)||(phase [xc] [yc] [zc]==0)){
			cumnum+=1;
		}
		}
	}
	return(cumnum);
}
```

```
/* routine to initialize system by determining local curvature */
/* of all phase 1 and phase 2 pixels */
/* Calls countem */
/* Called by runsint */
void sysinit(ph1,ph2)
        int ph1,ph2;
{
        int count,xl,yl,zl;

        count=0;
        /* process all pixels in the 3-D box */
        for(xl=0;xl<SYSSIZE;xl++){
        for(yl=0;yl<SYSSIZE;yl++){
        for(zl=0;zl<SYSSIZE;zl++){

                /* determine local curvature */
                /* For phase 1 want to determine number of porosity pixels */
                /* (phase=0) in immediate neighborhood */
                if(phase [xl] [yl] [zl]==ph1){
                        count=countem(xl,yl,zl,0);
                }
                /* For phase 2 want to determine number of porosity or phase */
                /* 2 pixels in immediate neighborhood */
                if(phase [xl] [yl] [zl]==ph2){
                        count=countem(xl,yl,zl,ph2);
                }
                if((count<0)||(count>=nsph)){
                        printf("Error count is %d \n",count);
                        printf("xl %d  yl %d  zl %d \n",xl,yl,zl);
                }

                /* case where we have a phase 1 surface pixel */
                /* with non-zero local curvature */
                if((count>=0)&&(phase [xl] [yl] [zl]==ph1)){

                        curvature [xl] [yl] [zl]=count;
                        /* update solid curvature histogram */
                        nsolid[count]+=1;
                }

                /* case where we have a phase 2 surface pixel */
                if((count>=0)&&(phase [xl] [yl] [zl]==ph2)){

                        curvature [xl] [yl] [zl]=count;
                        /* update air curvature histogram */
                        nair[count]+=1;
                }
```

```
          }
          }
          } /* end of xl loop */
}

/* routine to scan system and determine nsolid (ph2) and nair (ph1) */
/* histograms based on values in phase and curvature arrays */
/* Calls no other routines */
/* Called by runsint */
void sysscan(ph1,ph2)
        int ph1,ph2;
{
        int xd,yd,zd,curvval;

        /* Scan all pixels in 3-D system */
        for(xd=0;xd<SYSSIZE;xd++){
        for(yd=0;yd<SYSSIZE;yd++){
        for(zd=0;zd<SYSSIZE;zd++){

                curvval=curvature [xd] [yd] [zd];

                if(phase [xd] [yd] [zd]==ph2){
                        nair[curvval]+=1;
                }
                else if (phase [xd] [yd] [zd]==ph1){
                        nsolid[curvval]+=1;
                }
        }
        }
        }
}

/* routine to return how many cells of solid curvature histogram to use */
/* to accomodate nsearch pixels moving */
/* want to use highest values first */
/* Calls no other routines */
/* Called by movepix */
int procsol(nsearch)
        int nsearch;
{
        int valfound,i,stop;
        long int nsofar;

        /* search histogram from top down until cumulative count */
        /* exceeds nsearch */
        valfound=nsph-1;
        nsofar=0;
```

```
        stop=0;
        for(i=(nsph-1);((i>=0)&&(stop==0));i--){
                nsofar+=nsolid[i];
                if(nsofar>nsearch){
                        valfound=i;
                        stop=1;
                }
        }
        return(valfound);
}

/* routine to determine how many cells of air curvature histogram to use */
/* to accomodate nsearch moving pixels */
/* want to use lowest values first */
/* Calls no other routines */
/* Called by movepix */
int procair(nsearch)
        int nsearch;
{
        int valfound,i,stop;
        long int nsofar;

        /* search histogram from bottom up until cumulative count */
        /* exceeds nsearch */
        valfound=0;
        nsofar=0;
        stop=0;
        for(i=0;((i<nsph)&&(stop==0));i++){
                nsofar+=nair[i];
                if(nsofar>nsearch){
                        valfound=i;
                        stop=1;
                }
        }
        return(valfound);
}

/* routine to move requested number of pixels (ntomove) from highest */
/* curvature phase 1 (ph1) sites to lowest curvature phase 2 (ph2) sites */
/* Calls procsol and procair */
/* Called by runsint */
int movepix(ntomove,ph1,ph2)
        int ntomove,ph1,ph2;
{
        int xloc[2100],yloc[2100],zloc[2100];
        int count1,count2,ntot,countc,i,xp,yp,zp;
        int cmin,cmax,cfg;
```

```
int alldone;
long int nsolc,nairc,nsum,nsolm,nairm,nst1,nst2,next1,next2;
float pck,plsol,plair;

alldone=0;
/* determine critical values for removal and placement */
count1=procsol(ntomove);
nsum=0;
cfg=0;
cmax=count1;
for(i=nsph;i>count1;i--){
        if((nsolid[i]>0)&&(cfg==0)){
                cfg=1;
                cmax=i;
        }
        nsum+=nsolid[i];
}
/* Determine movement probability for last cell */
plsol=(float)(ntomove-nsum)/(float)nsolid[count1];
next1=ntomove-nsum;
nst1=nsolid[count1];

count2=procair(ntomove);
nsum=0;
cmin=count2;
cfg=0;
for(i=0;i<count2;i++){
        if((nair[i]>0)&&(cfg==0)){
                cfg=1;
                cmin=i;
        }
        nsum+=nair[i];
}
/* Determine movement probability for last cell */
plair=(float)(ntomove-nsum)/(float)nair[count2];
next2=ntomove-nsum;
nst2=nair[count2];

/* Check to see if equilibrium has been reached --- */
/* no further increase in hydraulic radius is possible */
if(cmin>=cmax){
        alldone=1;
        printf("Stopping - at equilibrium \n");
        printf("cmin- %d  cmax- %d \n",cmin,cmax);
        return(alldone);
}
```

```
/* initialize counters for performing sintering */
ntot=0;
nsolc=0;
nairc=0;
nsolm=0;
nairm=0;

/* Now process each pixel in turn */
for(xp=0;xp<SYSSIZE;xp++){
for(yp=0;yp<SYSSIZE;yp++){
for(zp=0;zp<SYSSIZE;zp++){

        countc=curvature [xp] [yp] [zp];
        /* handle phase 1 case first */
        if(phase [xp] [yp] [zp]==ph1){
                if(countc>count1){
                        /* convert from phase 1 to phase 2 */
                        phase [xp] [yp] [zp]=ph2;

                        /* update appropriate histogram cells */
                        nsolid[countc]-=1;
                        nair[countc]+=1;
                        /* store the location of the modified pixel */
                        ntot+=1;
                        xloc[ntot]=xp;
                        yloc[ntot]=yp;
                        zloc[ntot]=zp;
                }
                if(countc==count1){
                        nsolm+=1;
                     /* generate probability for pixel being removed */
                        pck=ran1(seed);
                        if((pck<0)||(pck>1.0)){pck=1.0;}

         if(((pck<plsol)&&(nsolc<next1))||((nst1-nsolm)<(next1-nsolc))){
                                nsolc+=1;
                                /* convert phase 1 pixel to phase 2 */
                                phase [xp] [yp] [zp]=ph2;

                                /* update appropriate histogram cells */
                                nsolid[count1]-=1;
                                nair[count1]+=1;
                          /* store the location of the modified pixel */
                                ntot+=1;
                                xloc[ntot]=xp;
                                yloc[ntot]=yp;
                                zloc[ntot]=zp;
```

```
                    }
                    }
            }

            /* handle phase 2 case here */
            else if (phase [xp] [yp] [zp]==ph2){
                    if(countc<count2){
                            /* convert phase 2 pixel to phase 1 */
                            phase [xp] [yp] [zp]=ph1;

                            nsolid[countc]+=1;
                            nair[countc]-=1;
                            ntot+=1;
                            xloc[ntot]=xp;
                            yloc[ntot]=yp;
                            zloc[ntot]=zp;
                    }
                    if(countc==count2){
                            nairm+=1;
                            pck=ran1(seed);
                            if((pck<0)||(pck>1.0)){pck=1.0;}

        if(((pck<plair)&&(nairc<next2))||((nst2-nairm)<(next2-nairc))){
                                    nairc+=1;
                                    /* convert phase 2 to phase 1 */
                                    phase [xp] [yp] [zp]=ph1;

                                    nsolid[count2]+=1;
                                    nair[count2]-=1;
                                    ntot+=1;
                                    xloc[ntot]=xp;
                                    yloc[ntot]=yp;
                                    zloc[ntot]=zp;
                            }
                    }
            }

    } /* end of zp loop */
    } /* end of yp loop */
    } /* end of xloop */
    printf("ntot is %d \n",ntot);
    return(alldone);
}

/* routine to execute user input number of cycles of sintering algorithm */
/* Calls maketemp, rhcalc, sysinit, sysscan, and movepix */
/* Called by main routine */
```

```
void runsint()
{
        int natonce,ncyc,i,rade,j,rflag;
        int ph1id,ph2id,keepgo;
        long int curvsum1,curvsum2,pixsum1,pixsum2;
        float rhnow,rhtarget,avecurv1,avecurv2;

        /* initialize the solid and air count histograms */
        for(i=0;i<=499;i++){
                nsolid[i]=0;
                nair[i]=0;
        }

        /* Obtain needed user input */
        printf("Enter phases to execute sintering between \n");
        scanf("%d %d",&ph1id,&ph2id);
        printf("%d %d \n",ph1id,ph2id);
        printf("Enter number of pixels to move at once (e.g. 200)\n");
        scanf("%d",&natonce);
        printf("%d \n",natonce);
        printf("Enter target hydraulic radius \n");
        scanf("%f",&rhtarget);
        printf("%f \n",rhtarget);
printf("Enter sphere radius to use for surface energy calculation (e.g. 3)\n");
        scanf("%d",&rade);
        printf("%d \n",rade);
        rflag=0;   /* always initialize system */

        nsph=maketemp(rade);
        printf("nsph is %d \n",nsph);
        if(rflag==0){
                sysinit(ph1id,ph2id);
        }
        else{
                sysscan(ph1id,ph2id);
        }
        i=0;
        rhnow=rhcalc(ph1id);
        while((rhnow<rhtarget)&&(i<MAXCYC)){
                printf("Now: %f  Target: %f \n",rhnow,rhtarget);
                i+=1;
                printf("Cycle: %d \n",i);
                keepgo=movepix(natonce,ph1id,ph2id);
                /* If equilibrium is reached, then return to calling routine */
                if(keepgo==1){
                        return;
                }
```

```
                curvsum1=0;
                curvsum2=0;
                pixsum1=0;
                pixsum2=0;
                /* Determine average curvatures for phases 1 and 2 */
                for(j=0;j<=nsph;j++){
                        pixsum1+=nsolid[j];
                        curvsum1+=(j*nsolid[j]);
                        pixsum2+=nair[j];
                        curvsum2+=(j*nair[j]);
                }
                avecurv1=(float)curvsum1/(float)pixsum1;
                avecurv2=(float)curvsum2/(float)pixsum2;
                printf("Ave. solid curvature: %f \n",avecurv1);
                printf("Ave. air curvature: %f \n",avecurv2);
                rhnow=rhcalc(ph1id);
        }
}

/* routine to read in microstructure from a file */
/* Calls no other routines */
/* Called by main routine */
void readmic()
{
        FILE *infile;
        char filen[80];
        int iout,i1,i2,i3;
        long int porcnt,totcnt;

        porcnt=totcnt=0;
        printf("Enter name of file to read in \n");
        scanf("%s",filen);
        infile=fopen(filen,"r");

        for(i1=0;i1<SYSSIZE;i1++){
        for(i2=0;i2<SYSSIZE;i2++){
        for(i3=0;i3<SYSSIZE;i3++){
                fscanf(infile,"%d",&iout);
                totcnt+=1;
                phase [i1] [i2] [i3]=iout;
                if(iout==0){
                        porcnt+=1;
                }
        }
        }
        }
        printf("Count for porosity is %ld out of %ld \n",porcnt,totcnt);
```

```
        fclose(infile);
}

/* routine to output microstructure to file */
/* Calls no other routines */
/* Called by main routine */
void savemic()
{
        FILE *outfile;
        int iout,i1,i2,i3;
        long int porcnt,totcnt;

        porcnt=totcnt=0;
        outfile=fopen("sintmic.img","w");

        for(i1=0;i1<SYSSIZE;i1++){
        for(i2=0;i2<SYSSIZE;i2++){
        for(i3=0;i3<SYSSIZE;i3++){
                totcnt+=1;
                iout=phase [i1] [i2] [i3];
                if(iout==0){
                        porcnt+=1;
                }
                fprintf(outfile,"%d\n",iout);
        }
        }
        }
        printf("Count for porosity is %ld out of %ld \n",porcnt,totcnt);
        fclose(outfile);
}

/* Calls phcount, runsint, readmic, and savemic */
main()
{
        int nseed,menuch;

        printf("Enter random number seed (integer < 0): \n");
        scanf("%d",&nseed);
        printf("%d \n",nseed);
        seed=(&nseed);

        menuch=5;
        /* Present menu and obtain user choice */
        while(menuch!=0){
                printf("Enter choice: \n");
                printf("0) Exit program \n");
                printf("1) Measure phase fractions \n");
```

```
            printf("2) Perform sintering algorithm \n");
            printf("3) Output resultant microstructure to file \n");
            printf("4) Read in microstructure from file \n");
            scanf("%d",&menuch);
            printf("%d \n",menuch);

            switch(menuch){
                    case 1:
                            phcount();
                            break;
                    case 2:
                            runsint();
                            break;
                    case 3:
                            savemic();
                            break;
                    case 4:
                            readmic();
                            break;
                    default:
                            break;
            }
    }
}
```

C Computer programs for three-dimensional cement hydration model

C.1 Code for assessing percolation of pore space

```
#define BURNT 70      /* label for a burnt pixel */
#define SIZE2D 49000 /* size of matrices for holding burning locations */
/* functions defining coordinates for burning in any of three directions */
#define cx(x,y,z,a,b,c) (1-b-c)*x+(1-a-c)*y+(1-a-b)*z
#define cy(x,y,z,a,b,c) (1-a-b)*x+(1-b-c)*y+(1-a-c)*z
#define cz(x,y,z,a,b,c) (1-a-c)*x+(1-a-b)*y+(1-b-c)*z

/* routine to assess the connectivity (percolation) of a single phase */
/* Two matrices are used here: one to store the recently burnt locations */
/*                the other to store the newly found burnt locations */
void burn3d(npix,d1,d2,d3)
        int npix;    /* ID of phase to perform burning on */
        int d1,d2,d3;   /* directional flags */
{
        long int ntop,nthrough,ncur,nnew,ntot;
        int i,inew,j,k,nmatx[SIZE2D],nmaty[SIZE2D],nmatz[SIZE2D];
        int xl,xh,j1,k1,px,py,pz,qx,qy,qz,xcn,ycn,zcn;
        int x1,y1,z1,igood,nnewx[SIZE2D],nnewy[SIZE2D],nnewz[SIZE2D];
        int jnew,icur;

/* counters for number of pixels of phase accessible from surface #1 */
/* and number which are part of a percolated pathway to surface #2 */
        ntop=0;
        nthrough=0;

        /* percolation is assessed from top to bottom only */
        /* and burning algorithm is periodic in other two directions */
        /* use of directional flags allow transformation of coordinates */
        /* to burn in direction of choosing (x, y, or z) */
        i=0;

        for(k=0;k<SYSIZE;k++){
        for(j=0;j<SYSIZE;j++){

                igood=0;
                ncur=0;
                ntot=0;
                /* Transform coordinates */
                px=cx(i,j,k,d1,d2,d3);
                py=cy(i,j,k,d1,d2,d3);
                pz=cz(i,j,k,d1,d2,d3);
```

```
                    if(mic [px] [py] [pz]==npix){
                        /* Start a burn front */
                        mic [px] [py] [pz]=BURNT;
                        ntot+=1;
                        ncur+=1;
                        /* burn front is stored in matrices nmat* */
                        /* and nnew* */
                        nmatx[ncur]=i;
                        nmaty[ncur]=j;
                        nmatz[ncur]=k;
                        /* Burn as long as new (fuel) pixels are found */
                        do{
                            nnew=0;
                            for(inew=1;inew<=ncur;inew++){
                                xcn=nmatx[inew];
                                ycn=nmaty[inew];
                                zcn=nmatz[inew];

                                /* Check all six neighbors */
                                for(jnew=1;jnew<=6;jnew++){
                                    x1=xcn;
                                    y1=ycn;
                                    z1=zcn;
                                    if(jnew==1){x1-=1;}
                                    if(jnew==2){x1+=1;}
                                    if(jnew==3){y1-=1;}
                                    if(jnew==4){y1+=1;}
                                    if(jnew==5){z1-=1;}
                                    if(jnew==6){z1+=1;}
/* Periodic in y and */
                                    if(y1>=SYSIZE){y1-=SYSIZE;}
                                    else if(y1<0){y1+=SYSIZE;}
/* Periodic in z direction */
                                    if(z1>=SYSIZE){z1-=SYSIZE;}
                                    else if(z1<0){z1+=SYSIZE;}

/* Nonperiodic so be sure to remain in the 3-D box */
                                    if((x1>=0)&&(x1<SYSIZE)){
                                    /* Transform coordinates */
                                    px=cx(x1,y1,z1,d1,d2,d3);
                                    py=cy(x1,y1,z1,d1,d2,d3);
                                    pz=cz(x1,y1,z1,d1,d2,d3);
                                    if(mic [px] [py] [pz]==npix){
                                      ntot+=1;
                                      mic [px] [py] [pz]=BURNT;
                                      nnew+=1;
                                      if(nnew>=SIZE2D){
```

```
                                printf("error in size of nnew \n");
                                        }
                                        nnewx[nnew]=x1;
                                        nnewy[nnew]=y1;
                                        nnewz[nnew]=z1;
                                    }
                        }
                        }
                }
                if(nnew>0){
                        ncur=nnew;
                        /* update the burn front matrices */
                        for(icur=1;icur<=ncur;icur++){
                                nmatx[icur]=nnewx[icur];
                                nmaty[icur]=nnewy[icur];
                                nmatz[icur]=nnewz[icur];
                        }
                }
        }while (nnew>0);

        ntop+=ntot;
        xl=0;
        xh=SYSIZE-1;
  /* See if current path extends through the microstructure */
        for(j1=0;j1<SYSIZE;j1++){
        for(k1=0;k1<SYSIZE;k1++){
                px=cx(xl,j1,k1,d1,d2,d3);
                py=cy(xl,j1,k1,d1,d2,d3);
                pz=cz(xl,j1,k1,d1,d2,d3);
                qx=cx(xh,j1,k1,d1,d2,d3);
                qy=cy(xh,j1,k1,d1,d2,d3);
                qz=cz(xh,j1,k1,d1,d2,d3);
if((mic [px] [py] [pz]==BURNT)&&(mic [qx] [qy] [qz]==BURNT)){
                        igood=2;
                }
                if(mic [px] [py] [pz]==BURNT){
                        mic [px] [py] [pz]=BURNT+1;
                }
                if(mic [qx] [qy] [qz]==BURNT){
                        mic [qx] [qy] [qz]=BURNT+1;
                }
        }
        }

        if(igood==2){
                nthrough+=ntot;
        }
```

```
                }
        }
        }

        printf("Phase ID= %d \n",npix);
        printf("Number accessible from first surface = %ld \n",ntop);
        printf("Number contained in through pathways= %ld \n",nthrough);

/* return the burnt sites to their original phase values */
        for(i=0;i<SYSIZE;i++){
        for(j=0;j<SYSIZE;j++){
        for(k=0;k<SYSIZE;k++){
                if(mic [i] [j] [k]>=BURNT){
                        mic [i] [j] [k]=npix;
                }
        }
        }
        }
}
```

C.2 Code for assessing percolation of total solids- set point

```
#define BURNT 70    /* label for burnt pixels */
#define SIZESET 100000
/* Transformation functions for changing direction of burn propagation */
#define cx(x,y,z,a,b,c) (1-b-c)*x+(1-a-c)*y+(1-a-b)*z
#define cy(x,y,z,a,b,c) (1-a-b)*x+(1-b-c)*y+(1-a-c)*z
#define cz(x,y,z,a,b,c) (1-a-c)*x+(1-a-b)*y+(1-b-c)*z

/* routine to assess connectivity (percolation) of solids for set estimation*/
/* Definition of set is a through pathway of cement particles connected */
/* together by CSH or ettringite */
/* Two matrices are used here: one to store the recently burnt locations */
/*                the other to store the newly found burnt locations */
int burnset(d1,d2,d3)
        int d1,d2,d3;
{
        long int ntop,nthrough,icur,inew,ncur,nnew,ntot;
        int i,j,k,setyet;
        static int nmatx[SIZESET],nmaty[SIZESET],nmatz[SIZESET];
        int xl,xh,j1,k1,px,py,pz,qx,qy,qz;
        int xcn,ycn,zcn,x1,y1,z1,igood;
        static int nnewx[SIZESET],nnewy[SIZESET],nnewz[SIZESET];
        int jnew;
        static char newmat [SYSIZE] [SYSIZE] [SYSIZE];
```

```
/* counters for number of pixels of phase accessible from surface #1 */
/* and number which are part of a percolated pathway to surface #2 */
        ntop=0;
        nthrough=0;
        setyet=0;
        for(k=0;k<SYSIZE;k++){
        for(j=0;j<SYSIZE;j++){
        for(i=0;i<SYSIZE;i++){
                newmat[i][j][k]=mic[i][j][k];
        }
        }
        }

        /* percolation is assessed from top to bottom only */
        /* in transformed coordinates */
        /* and burning algorithm is periodic in other two directions */
        i=0;

        for(k=0;k<SYSIZE;k++){
        for(j=0;j<SYSIZE;j++){

                igood=0;
                ncur=0;
                ntot=0;
                /* Transform coordinates */
                px=cx(i,j,k,d1,d2,d3);
                py=cy(i,j,k,d1,d2,d3);
                pz=cz(i,j,k,d1,d2,d3);
/* start from a cement clinker, ettringite, or CSH pixel */
         if((mic [px] [py] [pz]==C3S) ||
                (mic [px] [py] [pz]==C2S) ||
                (mic [px] [py] [pz]==CSH) ||
                (mic [px] [py] [pz]==ETTR) ||
                (mic [px] [py] [pz]==C3A) ||
                (mic [px] [py] [pz]==GYPSUM) ||
                (mic [px] [py] [pz]==C4AF)){
                        /* Start a burn front */
                        mic [px] [py] [pz]=BURNT;
                        ntot+=1;
                        ncur+=1;
                        /* burn front is stored in matrices nmat* */
                        /* and nnew* */
                        nmatx[ncur]=i;
                        nmaty[ncur]=j;
                        nmatz[ncur]=k;
                        /* Burn as long as new (fuel) pixels are found */
                        do{
```

```
                                nnew=0;
                                for(inew=1;inew<=ncur;inew++){
                                        xcn=nmatx[inew];
                                        ycn=nmaty[inew];
                                        zcn=nmatz[inew];
                                        /* Convert to directional coordinates */
                                        qx=cx(xcn,ycn,zcn,d1,d2,d3);
                                        qy=cy(xcn,ycn,zcn,d1,d2,d3);
                                        qz=cz(xcn,ycn,zcn,d1,d2,d3);

                                        /* Check all six neighbors */
                                        for(jnew=1;jnew<=6;jnew++){
                                                x1=xcn;
                                                y1=ycn;
                                                z1=zcn;
                                                if(jnew==1){x1-=1;}
                                                if(jnew==2){x1+=1;}
                                                if(jnew==3){y1-=1;}
                                                if(jnew==4){y1+=1;}
                                                if(jnew==5){z1-=1;}
                                                if(jnew==6){z1+=1;}
                                                /* Periodic in y and */
                                                if(y1>=SYSIZE){y1-=SYSIZE;}
                                                else if(y1<0){y1+=SYSIZE;}
                                                /* Periodic in z direction */
                                                if(z1>=SYSIZE){z1-=SYSIZE;}
                                                else if(z1<0){z1+=SYSIZE;}

/* Nonperiodic so be sure to remain in the 3-D box */
                                                if((x1>=0)&&(x1<SYSIZE)){
                                                px=cx(x1,y1,z1,d1,d2,d3);
                                                py=cy(x1,y1,z1,d1,d2,d3);
                                                pz=cz(x1,y1,z1,d1,d2,d3);
                /* Conditions for propagation of burning */
                /* 1) new pixel is CSH or ETTR */
                if((mic[px][py][pz]==CSH)||(mic[px][py][pz]==ETTR)){
                        ntot+=1;
                        mic [px] [py] [pz]=BURNT;
                        nnew+=1;
                        if(nnew>=SIZESET){
                              printf("error in size of nnew %d\n", nnew);
                        }
                        nnewx[nnew]=x1;
                        nnewy[nnew]=y1;
                        nnewz[nnew]=z1;
                }
/* 2) old pixel is CSH or ETTR and new pixel is one of cement clinker phases */
```

```
        else if(((newmat[qx][qy][qz]==CSH)||(newmat[qx][qy][qz]==ETTR))
        &&((mic [px] [py] [pz]==C3S) ||
        (mic [px] [py] [pz]==C2S) ||
        (mic [px] [py] [pz]==C3A) ||
        (mic [px] [py] [pz]==GYPSUM) ||
        (mic [px] [py] [pz]==C4AF))){
                        ntot+=1;
                        mic [px] [py] [pz]=BURNT;
                        nnew+=1;
                        if(nnew>=SIZESET){
                           printf("error in size of nnew %d\n", nnew);
                        }
                        nnewx[nnew]=x1;
                        nnewy[nnew]=y1;
                        nnewz[nnew]=z1;
        }
/* 3) old and new pixels belong to one of cement clinker phases and */
/* are contained in the same initial cement particle */
        else if((micpart[qx][qy][qz]==micpart[px][py][pz])
        &&((mic [px] [py] [pz]==C3S) ||
        (mic [px] [py] [pz]==C2S) ||
        (mic [px] [py] [pz]==C3A) ||
        (mic [px] [py] [pz]==GYPSUM) ||
        (mic [px] [py] [pz]==C4AF))&&((newmat[qx][qy][qz]==C3S)||
        (newmat [qx] [qy] [qz]==C2S) ||
        (newmat [qx] [qy] [qz]==C3A) ||
        (newmat [qx] [qy] [qz]==GYPSUM) ||
        (newmat[qx][qy][qz]==C4AF))){
                                ntot+=1;
                                mic [px] [py] [pz]=BURNT;
                                nnew+=1;
                                if(nnew>=SIZESET){
                        printf("error in size of nnew %d\n", nnew);
                                }
                                nnewx[nnew]=x1;
                                nnewy[nnew]=y1;
                                nnewz[nnew]=z1;
        }

                                } /* nonperiodic if delimiter */
                                } /* neighbors loop */
                        } /* propagators loop */
                        if(nnew>0){
                                ncur=nnew;
                                /* update the burn front matrices */
                                for(icur=1;icur<=ncur;icur++){
                                        nmatx[icur]=nnewx[icur];
                                        nmaty[icur]=nnewy[icur];
```

```
                                                nmatz[icur]=nnewz[icur];
                                        }
                                }
                        }while (nnew>0);

                        ntop+=ntot;
                        xl=0;
                        xh=SYSIZE-1;
                        /* Check for percolated path through system */
                        for(j1=0;j1<SYSIZE;j1++){
                        for(k1=0;k1<SYSIZE;k1++){
                                px=cx(xl,j1,k1,d1,d2,d3);
                                py=cy(xl,j1,k1,d1,d2,d3);
                                pz=cz(xl,j1,k1,d1,d2,d3);
                                qx=cx(xh,j1,k1,d1,d2,d3);
                                qy=cy(xh,j1,k1,d1,d2,d3);
                                qz=cz(xh,j1,k1,d1,d2,d3);
                if((mic [px] [py] [pz]==BURNT)&&(mic [qx] [qy] [qz]==BURNT)){
                                        igood=2;
                                }
                                if(mic [px] [py] [pz]==BURNT){
                                        mic [px] [py] [pz]=BURNT+1;
                                }
                                if(mic [qx] [qy] [qz]==BURNT){
                                        mic [qx] [qy] [qz]=BURNT+1;
                                }
                        }
                        }

                        if(igood==2){
                                nthrough+=ntot;
                        }
                }
        }
        }

        printf("Phase ID= Solid Phases \n");
        printf("Number accessible from first surface = %ld \n",ntop);
        printf("Number contained in through pathways= %ld \n",nthrough);
        if(nthrough>0){setyet=1;}

/* return the burnt sites to their original phase values */
        for(i=0;i<SYSIZE;i++){
        for(j=0;j<SYSIZE;j++){
        for(k=0;k<SYSIZE;k++){
                if(mic [i] [j] [k]>=BURNT){
                        mic [i] [j] [k]= newmat [i] [j] [k];
```

```
                }
        }
        }
        }
        /* Return flag indicating if set has indeed occurred */
        return(setyet);
}
```

C.3 Code for random number generation

```
/* Random number generator ran1 from Computers in Physics */
/* Volume 6 No. 5, 1992, 522-524, Press and Teukolsky */
/* To generate real random numbers 0.0-1.0 */
/* Should be seeded with a negative integer */
#define IA 16807
#define IM 2147483647
#define IQ 127773
#define IR 2836
#define NTAB 32
#define EPS (1.2E-07)
#define MAX(a,b) (a>b)?a:b
#define MIN(a,b) (a<b)?a:b

double ran1(idum)
int *idum;
{
        int j,k;
        static int iv[NTAB],iy=0;
        void nrerror();
        static double NDIV = 1.0/(1.0+(IM-1.0)/NTAB);
        static double RNMX = (1.0-EPS);
        static double AM = (1.0/IM);

        if ((*idum <= 0) || (iy == 0)) {
                *idum = MAX(-*idum,*idum);
                for(j=NTAB+7;j>=0;j--) {
                        k = *idum/IQ;
                        *idum = IA*(*idum-k*IQ)-IR*k;
                        if(*idum < 0) *idum += IM;
                        if(j < NTAB) iv[j] = *idum;
                }
                iy = iv[0];
        }
        k = *idum/IQ;
        *idum = IA*(*idum-k*IQ)-IR*k;
        if(*idum<0) *idum += IM;
```

```
        j = iy*NDIV;
        iy = iv[j];
        iv[j] = *idum;
        return MIN(AM*iy,RNMX);
}
#undef IA
#undef IM
#undef IQ
#undef IR
#undef NTAB
#undef EPS
#undef MAX
#undef MIN
```

C.4 Listing for hydreal3d.c

```
#define AGRATE 0.25        /* Probability of gypsum absorption by CSH */

/* routine to select a new neighboring location to (xloc, yloc, zloc) */
/* for a diffusing species */
/* Returns a prime number flag indicating direction chosen */
/* Calls ran1 */
/* Called by movecsh, extettr, extfh3, movegyp, extafm, moveettr, */
/* extpozz, movefh3, movech, extc3ah6, movec3a */
int moveone(xloc,yloc,zloc,act,sumold)
        int *xloc,*yloc,*zloc,*act,sumold;
{
        int plok,sumnew,xl1,yl1,zl1,act1;

        sumnew=1;
        /* store the input values for location */
        xl1=(*xloc);
        yl1=(*yloc);
        zl1=(*zloc);
        act1=(*act);

        /* Choose one of six directions (at random) for the new */
        /* location */
        plok=6.*ran1(seed);
        if((plok>5)||(plok<0)){plok=5;}

        switch (plok){
                case 0:
                        xl1-=1;
                        act1=1;
                        if(xl1<0){xl1=(SYSIZE-1);}
```

```
                        if(sumold%2!=0){sumnew=2;}
                        break;
                case 1:
                        xl1+=1;
                        act1=2;
                        if(xl1>=SYSIZE){xl1=0;}
                        if(sumold%3!=0){sumnew=3;}
                        break;
                case 2:
                        yl1-=1;
                        act1=3;
                        if(yl1<0){yl1=(SYSIZE-1);}
                        if(sumold%5!=0){sumnew=5;}
                        break;
                case 3:
                        yl1+=1;
                        act1=4;
                        if(yl1>=SYSIZE){yl1=0;}
                        if(sumold%7!=0){sumnew=7;}
                        break;
                case 4:
                        zl1-=1;
                        act1=5;
                        if(zl1<0){zl1=(SYSIZE-1);}
                        if(sumold%11!=0){sumnew=11;}
                        break;
                case 5:
                        zl1+=1;
                        act1=6;
                        if(zl1>=SYSIZE){zl1=0;}
                        if(sumold%13!=0){sumnew=13;}
                        break;
                default:
                        break;
        }

        /* Return the new location */
        *xloc=xl1;
        *yloc=yl1;
        *zloc=zl1;
        *act=act1;
        /* sumnew returns a prime number indicating that a specific direction */
        /* has been chosen */
        return(sumnew);
}

/* routine to move a diffusing CSH species */
```

```
/* Inputs: current location (xcur,ycur,zcur) and flag indicating if final */
/* step in diffusion process */
/* Returns flag indicating action taken (reaction or diffusion/no movement) */
/* Calls moveone */
/* Called by hydrate */
int movecsh(xcur,ycur,zcur,finalstep)
        int xcur,ycur,zcur,finalstep;
{
        int xnew,ynew,znew,plok,action,sumback,sumin,check;

        action=0;
        /* Store current location of species */
        xnew=xcur;
        ynew=ycur;
        znew=zcur;
        sumin=1;
        sumback=moveone(&xnew,&ynew,&znew,&action,sumin);

        if(action==0){printf("Error in value of action \n");}
        check=mic[xnew][ynew][znew];

/* if new location is solid C3S, C2S, or CSH, then convert */
/* diffusing CSH species to solid CSH */
        if((check==C3S)||(check==C2S)||(check==CSH)||(finalstep==1)){
                mic[xcur][ycur][zcur]=CSH;
                /* decrement count of diffusing CSH species */
                /* and increment count of solid CSH */
                count[DIFFCSH]-=1;
                count[CSH]+=1;
                action=0;
        }

        if(action!=0){
        /* if diffusion step is possible, perform it */
                if(check==POROSITY){
                        mic[xcur][ycur][zcur]=POROSITY;
                        mic[xnew][ynew][znew]=DIFFCSH;
                }
                else{
                        /* indicate that diffusing CSH species remained */
                        /* at original location */
                        action=7;
                }
        }
        return(action);
}
```

```
/* routine to return count of number of neighboring pixels for pixel */
/* (xck,yck,zck) which are not phase ph1, ph2, or ph3 which are input as */
/* parameters */
/* Calls no other routines */
/* Called by extettr, extfh3, extch, extafm, extpozz, extc3ah6 */
int edgecnt(xck,yck,zck,ph1,ph2,ph3)
        int xck,yck,zck,ph1,ph2,ph3;
{
        int ixe,iye,ize,edgeback,x2,y2,z2,check;

/* counter for number of neighboring pixels which are not ph1, ph2, or ph3 */
        edgeback=0;

/* Examine all pixels in a 3*3*3 box centered at (xck,yck,zck) */
/* except for the central pixel */
        for(ixe=(-1);ixe<=1;ixe++){
                x2=xck+ixe;
        for(iye=(-1);iye<=1;iye++){
                y2=yck+iye;
        for(ize=(-1);ize<=1;ize++){

                if((ixe!=0)||(iye!=0)||(ize!=0)){
                        z2=zck+ize;
                        /* adjust to maintain periodic boundaries */
                        if(x2<0){x2=(SYSIZE-1);}
                        else if(x2>=SYSIZE){x2=0;}
                        if(y2<0){y2=(SYSIZE-1);}
                        else if(y2>=SYSIZE){y2=0;}
                        if(z2<0){z2=(SYSIZE-1);}
                        else if(z2>=SYSIZE){z2=0;}
                        check=mic[x2][y2][z2];
                        if((check!=ph1)&&(check!=ph2)&&(check!=ph3)){
                                edgeback+=1;
                        }
                }
        }
        }
        }
        /* return number of neighboring pixels which are not ph1, ph2, or ph3 */
        return(edgeback);
}


/* routine to add extra ettringite when gypsum reacts with */
/* aluminates addition adjacent to location (xpres,ypres,zpres) */
/* in a fashion to preserve needle growth */
/* Returns flag indicating action taken */
/* Calls moveone and edgecnt */
```

```
/* Called by movegyp and movec3a */
int extettr(xpres,ypres,zpres)
        int xpres,ypres,zpres;
{
        int check,newact,multf,numnear,sump,xchr,ychr,zchr,fchr,i1,plok;
        float pneigh,ptest;
        long int tries;

/* first try 6 neighboring locations until      */
/* a) successful                                */
/* b) all 6 sites are tried and full or         */
/* c) 500 tries are made                        */
        fchr=0;
        sump=1;
        /* Note that 30030 = 2*3*5*7*11*13 */
/* indicating that all six sites have been tried */
        for(i1=1;((i1<=500)&&(fchr==0)&&(sump!=30030));i1++){

/* determine location of neighbor (using periodic boundaries) */
                xchr=xpres;
                ychr=ypres;
                zchr=zpres;
                newact=0;
                multf=moveone(&xchr,&ychr,&zchr,&newact,sump);
                if(newact==0){printf("Error in value of action \n");}

                check=mic[xchr][ychr][zchr];

                /* if neighbor is porosity, and conditions are favorable */
                /* based on number of neighboring ettringite, C3A, or C4AF */
                /* pixels then locate the ettringite there */
                if(check==POROSITY){
                        numnear=edgecnt(xchr,ychr,zchr,ETTR,C3A,C4AF);
                        pneigh=(float)(numnear+1)/26.0;
                        pneigh*=pneigh;
                        ptest=ran1(seed);
                        if(pneigh>=ptest){
                                mic[xchr][ychr][zchr]=ETTR;
                                fchr=1;
                        }
                }
                else{
                        sump*=multf;
                }
        }

/* if no neighbor available, locate ettringite at random location */
```

```
/* in pore space in contact with at least another ettringite */
/* or aluminate surface  */
        tries=0;
        while(fchr==0){
                tries+=1;
                newact=7;
                /* generate a random location in the 3-D system */
                xchr=(int)((float)SYSIZE*ran1(seed));
                ychr=(int)((float)SYSIZE*ran1(seed));
                zchr=(int)((float)SYSIZE*ran1(seed));
                if(xchr>=SYSIZE){xchr=0;}
                if(ychr>=SYSIZE){ychr=0;}
                if(zchr>=SYSIZE){zchr=0;}

                check=mic[xchr][ychr][zchr];
                /* if location is porosity, locate the ettringite there */
                if(check==POROSITY){
                        numnear=edgecnt(xchr,ychr,zchr,ETTR,C3A,C4AF);
                        /* be sure that at least one neighboring pixel */
                        /* is ettringite, or aluminate clinker */
                        if((tries>5000)||(numnear<26)){
                                mic[xchr][ychr][zchr]=ETTR;
                                fchr=1;
                        }
                }
        }
        return(newact);
}

/* routine to add extra FH3 when gypsum reacts with */
/* C4AF at location (xpres,ypres,zpres) */
/* Called by movegyp and moveettr */
/* Calls moveone and edgecnt */
void extfh3(xpres,ypres,zpres)
        int xpres,ypres,zpres;
{
        int multf,numnear,sump,xchr,ychr,zchr,check,fchr,i1,plok,newact;
        long int tries;

/* first try 6 neighboring locations until       */
/* a) successful                                */
/* b) all 6 sites are tried and full or         */
/* c) 500 tries are made                        */
        fchr=0;
        sump=1;
        for(i1=1;((i1<=500)&&(fchr==0)&&(sump!=30030));i1++){
```

```
                /* choose a neighbor at random */
                xchr=xpres;
                ychr=ypres;
                zchr=zpres;
                newact=0;
                multf=moveone(&xchr,&ychr,&zchr,&newact,sump);
                if(newact==0){printf("Error in value of newact in extfh3 \n");}
                check=mic[xchr][ychr][zchr];

                /* if neighbor is porosity   */
                /* then locate the FH3 there */
                if(check==POROSITY){
                        mic[xchr][ychr][zchr]=FH3;
                        fchr=1;
                }
                else{
                        sump*=multf;
                }
        }

/* if no neighbor available, locate FH3 at random location */
/* in pore space in contact with at least one FH3 */
        tries=0;
        while(fchr==0){
                tries+=1;
                /* generate a random location in the 3-D system */
                xchr=(int)((float)SYSIZE*ran1(seed));
                ychr=(int)((float)SYSIZE*ran1(seed));
                zchr=(int)((float)SYSIZE*ran1(seed));
                if(xchr>=SYSIZE){xchr=0;}
                if(ychr>=SYSIZE){ychr=0;}
                if(zchr>=SYSIZE){zchr=0;}
                check=mic[xchr][ychr][zchr];
                /* if location is porosity, locate the FH3 there */
                if(check==POROSITY){
                        numnear=edgecnt(xchr,ychr,zchr,FH3,FH3,DIFFFH3);
                        /* be sure that at least one neighboring pixel */
                        /* is FH3 or diffusing FH3 */
                        if((numnear<26)||(tries>5000)){
                                mic[xchr][ychr][zchr]=FH3;
                                fchr=1;
                        }
                }
        }
}

/* routine to add extra CH when gypsum reacts with */
```

```
/* C4AF */
/* Called by movegyp and moveettr */
/* Calls edgecnt */
void extch()
{
        int numnear,sump,xchr,ychr,zchr,fchr,i1,plok,check;
        long int tries;

        fchr=0;
        tries=0;
        /* locate CH at random location */
        /* in pore space in contact with at least another CH */
        while(fchr==0){
                tries+=1;
                /* generate a random location in the 3-D system */
                xchr=(int)((float)SYSIZE*ran1(seed));
                ychr=(int)((float)SYSIZE*ran1(seed));
                zchr=(int)((float)SYSIZE*ran1(seed));
                if(xchr>=SYSIZE){xchr=0;}
                if(ychr>=SYSIZE){ychr=0;}
                if(zchr>=SYSIZE){zchr=0;}
                check=mic[xchr][ychr][zchr];

                /* if location is porosity, locate the CH there */
                if(check==POROSITY){
                        numnear=edgecnt(xchr,ychr,zchr,CH,DIFFCH,CH);
                        /* be sure that at least one neighboring pixel */
                        /* is CH or diffusing CH */
                        if((numnear<26)||(tries>5000)){
                                mic[xchr][ychr][zchr]=CH;
                                fchr=1;
                        }
                }
        }
}

/* routine to move a diffusing gypsum species */
/* from current location (xcur,ycur,zcur) */
/* Returns action flag indicating response taken */
/* Called by hydrate */
/* Calls moveone, extettr, extch, and extfh3 */
int movegyp(xcur,ycur,zcur,finalstep)
        int xcur,ycur,zcur,finalstep;
{
        int check,xnew,ynew,znew,plok,action,nexp,iexp;
        int xexp,yexp,zexp,newact,sumold,sumgarb;
        float pexp,pext;
```

```
        sumold=1;

/* First be sure that a diffusing gypsum species is located at xcur,ycur,zcur */
/* if not, return to calling routine */
        if(mic[xcur][ycur][zcur]!=DIFFGYP){
                action=0;
                return(action);
        }

/* Determine new coordinates (periodic boundaries are used) */
        xnew=xcur;
        ynew=ycur;
        znew=zcur;
        action=0;
        sumgarb=moveone(&xnew,&ynew,&znew,&action,sumold);
        if(action==0){printf("Error in value of action in movegyp \n");}
        check=mic[xnew][ynew][znew];

        /* if new location is CSH, check for absorption of gypsum */
        if((check==CSH)&&((float)count[ABSGYP]<(gypabsprob*(float)count[CSH]))){
                pexp=ran1(seed);
                if(pexp<AGRATE){
                        /* update counts for absorbed and diffusing gypsum */
                        count[ABSGYP]+=1;
                        count[DIFFGYP]-=1;
                        mic[xcur][ycur][zcur]=ABSGYP;
                        action=0;
                }
        }

        /* if new location is C3A or diffusing C3A, execute conversion */
        /* to ettringite (including necessary volumetric expansion) */
        if((check==C3A)||(check==DIFFC3A)){
        /* Convert diffusing gypsum to an ettringite pixel */
                action=0;
                mic[xcur][ycur][zcur]=ETTR;
                count[DIFFGYP]-=1;
                if(check==DIFFC3A){
                /* decrement count of diffusing C3A species */
                        count[DIFFC3A]-=1;
                }

                /* determine if C3A should be converted to ettringite */
                /* 1 unit of gypsum requires 0.40 units of C3A */
                /* and should form 3.30 units of ettringite */
                pexp=ran1(seed);
                nexp=2;
```

```
		if(pexp<=0.40){
			mic[xnew][ynew][znew]=ETTR;
			nexp=1;
		}
		else{
			/* maybe someday, use a new FIXEDC3A here */
			/* so it won't dissolve later */
			if(check==C3A){
				mic[xnew][ynew][znew]=C3A;
			}
			else{
				count[DIFFC3A]+=1;
				mic[xnew][ynew][znew]=DIFFC3A;
			}
			nexp=2;
		}

/* create extra ettringite pixels to maintain volume stoichiometry */
/* xexp, yexp, and zexp hold coordinates of most recently added ettringite */
/* species as we attempt to grow a needle like structure */
		xexp=xcur;
		yexp=ycur;
		zexp=zcur;
		for(iexp=1;iexp<=nexp;iexp++){
			newact=extettr(xexp,yexp,zexp);
			/* update xexp, yexp and zexp as needed */
			switch (newact){
				case 1:
					xexp-=1;
					if(xexp<0){xexp=(SYSIZE-1);}
					break;
				case 2:
					xexp+=1;
					if(xexp>=SYSIZE){xexp=0;}
					break;
				case 3:
					yexp-=1;
					if(yexp<0){yexp=(SYSIZE-1);}
					break;
				case 4:
					yexp+=1;
					if(yexp>=SYSIZE){yexp=0;}
					break;
				case 5:
					zexp-=1;
					if(zexp<0){zexp=(SYSIZE-1);}
					break;
```

```
                                case 6:
                                        zexp+=1;
                                        if(zexp>=SYSIZE){zexp=0;}
                                        break;
                                default:
                                        break;
                        }
                }

                /* probabilistic-based expansion for last ettringite pixel */
                pexp=ran1(seed);
                if(pexp<=0.30){
                        newact=extettr(xexp,yexp,zexp);
                }
        }

        /* if new location is C4AF execute conversion */
        /* to ettringite (including necessary volumetric expansion) */
        if(check==C4AF){
                mic[xcur][ycur][zcur]=ETTR;
                count[DIFFGYP]-=1;

                /* determine if C4AF should be converted to ettringite */
                /* 1 unit of gypsum requires 0.575 units of C4AF */
                /* and should form 3.30 units of ettringite */
                pexp=ran1(seed);
                nexp=2;
                if(pexp<=0.575){
                        mic[xnew][ynew][znew]=ETTR;
                        nexp=1;
                        pext=ran1(seed);
                        /* Addition of extra CH */
                        if(pext<0.2584){
                                extch();
                        }
                        pext=ran1(seed);
                        /* Addition of extra FH3 */
                        if(pext<0.5453){
                                extfh3(xnew,ynew,znew);
                        }

                }
                else{
                        /* maybe someday, use a new FIXEDC4AF here */
                        /* so it won't dissolve later */
                        mic[xnew][ynew][znew]=C4AF;
                        nexp=2;
```

```
                }

/* create extra ettringite pixels to maintain volume stoichiometry */
/* xexp, yexp and zexp hold coordinates of most recently added ettringite */
/* species as we attempt to grow a needle like structure */
                xexp=xcur;
                yexp=ycur;
                zexp=zcur;
                for(iexp=1;iexp<=nexp;iexp++){
                        newact=extettr(xexp,yexp,zexp);
                        /* update xexp, yexp and zexp as needed */
                        switch (newact){
                                case 1:
                                        xexp-=1;
                                        if(xexp<0){xexp=(SYSIZE-1);}
                                        break;
                                case 2:
                                        xexp+=1;
                                        if(xexp>=SYSIZE){xexp=0;}
                                        break;
                                case 3:
                                        yexp-=1;
                                        if(yexp<0){yexp=(SYSIZE-1);}
                                        break;
                                case 4:
                                        yexp+=1;
                                        if(yexp>=SYSIZE){yexp=0;}
                                        break;
                                case 5:
                                        zexp-=1;
                                        if(zexp<0){zexp=(SYSIZE-1);}
                                        break;
                                case 6:
                                        zexp+=1;
                                        if(zexp>=SYSIZE){zexp=0;}
                                        break;
                                default:
                                        break;
                        }
                }

                /* probabilistic-based expansion for last ettringite pixel */
                pexp=ran1(seed);
                if(pexp<=0.30){
                        newact=extettr(xexp,yexp,zexp);
                }
                action=0;
```

```
        }

        /* if last diffusion step and no reaction, convert back to */
        /* solid gypsum */
        if((action!=0)&&(finalstep==1)){
                action=0;
                count[DIFFGYP]-=1;
                mic[xcur][ycur][zcur]=GYPSUM;
        }

        if(action!=0){
                /* if diffusion is possible, execute it */
                if(check==POROSITY){
                        mic[xcur][ycur][zcur]=POROSITY;
                        mic[xnew][ynew][znew]=DIFFGYP;
                }
                else{
                        /* indicate that diffusing gypsum remained at */
                        /* original location */
                        action=7;
                }
        }
        return(action);
}

/* routine to add extra AFm phase when diffusing ettringite reacts */
/* with C3A (diffusing or solid) at location (xpres,ypres,zpres) */
/* Called by moveettr and movec3a */
/* Calls moveone and edgecnt */
void extafm(xpres,ypres,zpres)
        int xpres,ypres,zpres;
{
        int check,sump,xchr,ychr,zchr,fchr,i1,plok,newact,numnear;
        long int tries;

/* first try 6 neighboring locations until      */
/* a) successful                               */
/* b) all 6 sites are tried or                 */
/* c) 100 tries are made                       */
        fchr=0;
        sump=1;
        for(i1=1;((i1<=100)&&(fchr==0)&&(sump!=30030));i1++){

                /* determine location of neighbor (using periodic boundaries) */
                xchr=xpres;
                ychr=ypres;
                zchr=zpres;
```

```
                newact=0;
                sump*=moveone(&xchr,&ychr,&zchr,&newact,sump);
                if(newact==0){printf("Error in value of newact in extafm \n");}
                check=mic[xchr][ychr][zchr];

                /* if neighbor is porosity, locate the AFm phase there */
                if(check==POROSITY){
                        mic[xchr][ychr][zchr]=AFM;
                        fchr=1;
                }
        }

        /* if no neighbor available, locate AFm phase at random location */
        /* in pore space */
        tries=0;
        while(fchr==0){
                tries+=1;
                /* generate a random location in the 3-D system */
                xchr=(int)((float)SYSIZE*ran1(seed));
                ychr=(int)((float)SYSIZE*ran1(seed));
                zchr=(int)((float)SYSIZE*ran1(seed));
                if(xchr>=SYSIZE){xchr=0;}
                if(ychr>=SYSIZE){ychr=0;}
                if(zchr>=SYSIZE){zchr=0;}
                check=mic[xchr][ychr][zchr];

                /* if location is porosity, locate the extra AFm there */
                if(check==POROSITY){
                        numnear=edgecnt(xchr,ychr,zchr,AFM,C3A,C4AF);
                        /* Be sure that at least one neighboring pixel is */
                        /* Afm phase, C3A, or C4AF */
                        if((tries>5000)||(numnear<26)){
                                mic[xchr][ychr][zchr]=AFM;
                                fchr=1;
                        }
                }
        }
}

/* routine to move a diffusing ettringite species */
/* currently located at (xcur,ycur,zcur) */
/* Called by hydrate */
/* Calls moveone, extch, extfh3, and extafm */
int moveettr(xcur,ycur,zcur,finalstep)
        int xcur,ycur,zcur,finalstep;
{
        int check,xnew,ynew,znew,plok,action,nexp,iexp;
```

```
        int xexp,yexp,zexp,newact,sumold,sumgarb;
        float pexp,pafm,pgrow;

/* First be sure a diffusing ettringite species is located at xcur,ycur,zcur */
/* if not, return to calling routine */
        if(mic[xcur][ycur][zcur]!=DIFFETTR){
                action=0;
                return(action);
        }

/* Determine new coordinates (periodic boundaries are used) */
        xnew=xcur;
        ynew=ycur;
        znew=zcur;
        action=0;
        sumold=1;
        sumgarb=moveone(&xnew,&ynew,&znew,&action,sumold);
        if(action==0){printf("Error in value of action in moveettr \n");}
        check=mic[xnew][ynew][znew];

        /* if new location is C4AF, execute conversion */
        /* to AFM phase (including necessary volumetric expansion) */
        if(check==C4AF){
                /* Convert diffusing ettringite to AFM phase */
                mic[xcur][ycur][zcur]=AFM;
                count[DIFFETTR]-=1;

                /* determine if C4AF should be converted to Afm */
                /* or FH3- 1 unit of ettringite requires 0.348 units */
                /* of C4AF to form 1.278 units of Afm, */
                /* 0.0901 units of CH and 0.1899 units of FH3 */
                pexp=ran1(seed);

                if(pexp<=0.278){
                        mic[xnew][ynew][znew]=AFM;
                        pafm=ran1(seed);
                        /* 0.3241= 0.0901/0.278 */
                        if(pafm<=0.3241){
                                extch();
                        }
                        pafm=ran1(seed);
                        /* 0.4313= ((.1899-(.348-.278))/.278)   */
                        if(pafm<=0.4313){
                                extfh3(xnew,ynew,znew);
                        }
                }
                else if (pexp<=0.348){
```

```
                        mic[xnew][ynew][znew]=FH3;
                }
                action=0;
        }

        /* if new location is C3A or diffusing C3A, execute conversion */
        /* to AFM phase (including necessary volumetric expansion) */
        if((check==C3A)||(check==DIFFC3A)){
                /* Convert diffusing ettringite to AFM phase */
                action=0;
                mic[xcur][ycur][zcur]=AFM;
                count[DIFFETTR]-=1;

                if(check==DIFFC3A){
                /* decrement count of diffusing C3A species */
                        count[DIFFC3A]-=1;
                }

                /* determine if C3A should be converted to AFm */
                /* 1 unit of ettringite requires 0.2424 units of C3A */
                /* and should form 1.278 units of AFm phase */
                pexp=ran1(seed);
                if(pexp<=0.2424){
                        mic[xnew][ynew][znew]=AFM;
                        pafm=(-0.1);
                }
                else{
                        /* maybe someday, use a new FIXEDC3A here */
                        /* so it won't dissolve later */
                        if(check==C3A){
                                mic[xnew][ynew][znew]=C3A;
                        }
                        else{
                                count[DIFFC3A]+=1;
                                mic[xnew][ynew][znew]=DIFFC3A;
                        }
                        pafm=(0.278-0.2424)/(1.0-0.2424);
                }

                /* probabilistic-based expansion for new AFm phase pixel */
                pexp=ran1(seed);
                if(pexp<=pafm){
                        extafm(xcur,ycur,zcur);
                }
        }

        /* Check for conversion back to solid ettringite */
```

```
        if(check==ETTR){
                pgrow=ran1(seed);
                if(pgrow<=ETTRGROW){
                        mic[xcur] [ycur] [zcur]=ETTR;
                        action=0;
                        count[DIFFETTR]-=1;
                }
        }

        /* if last diffusion step and no reaction, convert back to */
        /* solid ettringite */
        if((action!=0)&&(finalstep==1)){
                action=0;
                count[DIFFETTR]-=1;
                mic[xcur][ycur][zcur]=ETTR;
        }

        if(action!=0){
                /* if diffusion is possible, execute it */
                if(check==POROSITY){
                        mic[xcur][ycur][zcur]=POROSITY;
                        mic[xnew][ynew][znew]=DIFFETTR;
                }
                else{
                        /* indicate that diffusing ettringite remained at */
                        /* original location */
                        action=7;
                }
        }
        return(action);
}

/* routine to add extra pozzolanic CSH when CH reacts at */
/* pozzolanic surface (e.g. silica fume) located at (xpres,ypres,zpres) */
/* Called by movech */
/* Calls moveone and edgecnt */
void extpozz(xpres,ypres,zpres)
        int xpres,ypres,zpres;
{
        int check,sump,xchr,ychr,zchr,fchr,i1,plok,action,numnear;
        long int tries;

/* first try 6 neighboring locations until      */
/* a) successful                                */
/* b) all 6 sites are tried or                  */
/* c) 100 tries are made                        */
        fchr=0;
```

```
        sump=1;
        for(i1=1;((i1<=100)&&(fchr==0)&&(sump!=30030));i1++){

                /* determine location of neighbor (using periodic boundaries) */
                xchr=xpres;
                ychr=ypres;
                zchr=zpres;
                action=0;
                sump*=moveone(&xchr,&ychr,&zchr,&action,sump);
                if(action==0){printf("Error in value of action in extpozz \n");}
                check=mic[xchr][ychr][zchr];

                /* if neighbor is porosity, locate the pozzolanic CSH there */
                if(check==POROSITY){
                        mic[xchr][ychr][zchr]=POZZ;
                        fchr=1;
                }
        }

        /* if no neighbor available, locate pozzolanic CSH at random location */
        /* in pore space */
        tries=0;
        while(fchr==0){
                tries+=1;
                /* generate a random location in the 3-D system */
                xchr=(int)((float)SYSIZE*ran1(seed));
                ychr=(int)((float)SYSIZE*ran1(seed));
                zchr=(int)((float)SYSIZE*ran1(seed));
                if(xchr>=SYSIZE){xchr=0;}
                if(ychr>=SYSIZE){ychr=0;}
                if(zchr>=SYSIZE){zchr=0;}
                check=mic[xchr][ychr][zchr];
          /* if location is porosity, locate the extra pozzolanic CSH there */
                if(check==POROSITY){
                        numnear=edgecnt(xchr,ychr,zchr,POZZ,CSH,POZZ);
                        /* Be sure that one neighboring species is CSH or */
                        /* pozzolanic material */
                        if((tries>5000)||(numnear<26)){
                                mic[xchr][ychr][zchr]=POZZ;
                                fchr=1;
                        }
                }
        }
}


/* routine to move a diffusing FH3 species */
/* from location (xcur,ycur,zcur) with nucleation probability nucprob */
```

```
/* Called by hydrate */
/* Calls moveone */
int movefh3(xcur,ycur,zcur,finalstep,nucprob)
        int xcur,ycur,zcur,finalstep;
        float nucprob;
{
        int check,xnew,ynew,znew,plok,action,sumold,sumgarb;
        float pgen;

        /* first check for nucleation */
        pgen=ran1(seed);

        if((nucprob>=pgen)||(finalstep==1)){
                action=0;
                mic[xcur][ycur][zcur]=FH3;
                count[DIFFFH3]-=1;
        }
        else{

                /* determine new location (using periodic boundaries) */
                xnew=xcur;
                ynew=ycur;
                znew=zcur;
                action=0;
                sumold=1;
                sumgarb=moveone(&xnew,&ynew,&znew,&action,sumold);
                if(action==0){printf("Error in value of action in movefh3 \n");}
                check=mic[xnew][ynew][znew];

                /* check for growth of FH3 crystal */
                if(check==FH3){
                        mic[xcur][ycur][zcur]=FH3;
                        count[DIFFFH3]-=1;
                        action=0;
                }

                if(action!=0){
                        /* if diffusion is possible, execute it */
                        if(check==POROSITY){
                                mic[xcur][ycur][zcur]=POROSITY;
                                mic[xnew][ynew][znew]=DIFFFH3;
                        }
                        else{
                                /* indicate that diffusing FH3 species */
                                /* remained at original location */
                                action=7;
                        }
```

```
                }
        }
        return(action);
}

/* routine to move a diffusing CH species */
/* from location (xcur,ycur,zcur) with nucleation probability nucprob */
/* Called by hydrate */
/* Calls moveone and extpozz */
int movech(xcur,ycur,zcur,finalstep,nucprob)
        int xcur,ycur,zcur,finalstep;
        float nucprob;
{
        int check,xnew,ynew,znew,plok,action,sumgarb,sumold;
        float pexp,pgen;

        /* first check for nucleation */
        pgen=ran1(seed);
        if((nucprob>=pgen)||(finalstep==1)){
                action=0;
                mic[xcur][ycur][zcur]=CH;
                count[DIFFCH]-=1;
        }
        else{

                /* determine new location (using periodic boundaries) */
                xnew=xcur;
                ynew=ycur;
                znew=zcur;
                action=0;
                sumold=1;
                sumgarb=moveone(&xnew,&ynew,&znew,&action,sumold);
                if(action==0){printf("Error in value of action in movech \n");}
                check=mic[xnew][ynew][znew];

                /* check for growth of CH crystal */
                if(check==CH){
                        mic[xcur][ycur][zcur]=CH;
                        count[DIFFCH]-=1;
                        action=0;
                }

                /* check for pozzolanic reaction */
                /* 36.41 units of CH can react with 27 units of S */
                if((check==POZZ)&&(npr<=(int)((float)nfill*1.35))){
                        mic[xcur][ycur][zcur]=POZZ;
                        /* update counter of number of diffusing CH */
```

```
                /* which have reacted pozzolanically */
                npr+=1;
                count[DIFFCH]-=1;
                action=0;
                /* allow for extra pozzolanic CSH as needed */
                pexp=ran1(seed);
                /* 0.4795=(80.87-36.41-27)/36.41 */
                if(pexp<=0.4795){
                        extpozz(xcur,ycur,zcur);
                }
            }
            if(action!=0){
                /* if diffusion is possible, execute it */
                if(check==POROSITY){
                        mic[xcur][ycur][zcur]=POROSITY;
                        mic[xnew][ynew][znew]=DIFFCH;
                }
                else{
                        /* indicate that diffusing CH species */
                        /* remained at original location */
                        action=7;
                }
            }
        }
        return(action);
}

/* routine to add extra C3AH6 when diffusing C3A nucleates or reacts at */
/* C3AH6 surface at location (xpres,ypres,zpres) */
/* Called by movec3a */
/* Calls moveone and edgecnt */
void extc3ah6(xpres,ypres,zpres)
        int xpres,ypres,zpres;
{
        int check,sump,xchr,ychr,zchr,fchr,i1,plok,action,numnear;
        long int tries;

/* First try 6 neighboring locations until      */
/*  a) successful                               */
/* b) all 6 sites are tried or                  */
/* c) 100 random attempts are made              */
        fchr=0;
        sump=1;
        for(i1=1;((i1<=100)&&(fchr==0)&&(sump!=30030));i1++){

                /* determine new coordinates (using periodic boundaries) */
                xchr=xpres;
```

```
                ychr=ypres;
                zchr=zpres;
                action=0;
                sump*=moveone(&xchr,&ychr,&zchr,&action,sump);
                if(action==0){printf("Error in action value in extc3ah6 \n");}
                check=mic[xchr][ychr][zchr];

                /* if neighbor is pore space, convert it to C3AH6 */
                if(check==POROSITY){
                        mic[xchr][ychr][zchr]=C3AH6;
                        fchr=1;
                }
        }

        /* if unsuccessful, add C3AH6 at random location in pore space */
        tries=0;
        while(fchr==0){
                tries+=1;
                xchr=(int)((float)SYSIZE*ran1(seed));
                ychr=(int)((float)SYSIZE*ran1(seed));
                zchr=(int)((float)SYSIZE*ran1(seed));
                if(xchr>=SYSIZE){xchr=0;}
                if(ychr>=SYSIZE){ychr=0;}
                if(zchr>=SYSIZE){zchr=0;}
                check=mic[xchr][ychr][zchr];

                if(check==POROSITY){
                        numnear=edgecnt(xchr,ychr,zchr,C3AH6,C3A,C3AH6);
                        /* Be sure that new C3AH6 is in contact with */
                        /* at least one C3AH6 or C3A */
                        if((tries>5000)||(numnear<26)){
                                mic[xchr][ychr][zchr]=C3AH6;
                                fchr=1;
                        }
                }
        }
}

/* routine to move a diffusing C3A species */
/* from location (xcur,ycur,zcur) with nucleation probability of nucprob */
/* Called by hydrate */
/* Calls extc3ah6, moveone, extettr, and extafm */
int movec3a(xcur,ycur,zcur,finalstep,nucprob)
        int xcur,ycur,zcur,finalstep;
        float nucprob;
{
        int check,xnew,ynew,znew,plok,action,sumgarb,sumold;
```

```
int xexp,yexp,zexp,nexp,iexp,newact;
float pgen,pexp,pafm,pgrow;

/* First be sure that a diffusing C3A species is at (xcur,ycur,zcur) */
if(mic[xcur][ycur][zcur]!=DIFFC3A){
        action=0;
        return(action);
}

/* Check for nucleation into solid C3AH6 */
pgen=ran1(seed);

if((nucprob>=pgen)||(finalstep==1)){
        action=0;
        mic[xcur][ycur][zcur]=C3AH6;
        /* decrement count of diffusing C3A species */
        count[DIFFC3A]-=1;
        /* allow for probabilistic-based expansion of C3AH6 */
        /* crystal to account for volume stoichiometry */
        pexp=ran1(seed);
        if(pexp<=0.69){
                extc3ah6(xcur,ycur,zcur);
        }
}
else{
        /* determine new coordinates (using periodic boundaries) */
        xnew=xcur;
        ynew=ycur;
        znew=zcur;
        action=0;
        sumold=1;
        sumgarb=moveone(&xnew,&ynew,&znew,&action,sumold);
        if(action==0){printf("Error in value of action in movec3a \n");}
        check=mic[xnew][ynew][znew];

        /* check for growth of C3AH6 crystal */
        if(check==C3AH6){
        pgrow=ran1(seed);
                /* Try to slow down growth of C3AH6 crystals to */
                /* promote ettringite and Afm formation */
                if(pgrow<=C3AH6GROW){
                        mic[xcur][ycur][zcur]=C3AH6;
                        count[DIFFC3A]-=1;
                        action=0;
                 /* allow for probabilistic-based expansion of C3AH6 */
                 /* crystal to account for volume stoichiometry */
                        pexp=ran1(seed);
```

```
                        if(pexp<=0.69){
                                extc3ah6(xcur,ycur,zcur);
                        }
                }
        }

        /* examine reaction with diffusing gypsum to form ettringite */
        /* Only allow reaction with diffusing gypsum */
        if(check==DIFFGYP){
                /* convert diffusing gypsum to ettringite */
                mic[xnew][ynew][znew]=ETTR;
                /* decrement counts of diffusing C3A and gypsum */
                count[DIFFC3A]-=1;
                count[DIFFGYP]-=1;
                action=0;

/* convert diffusing C3A to solid ettringite or else leave as a diffusing C3A */
                pexp=ran1(seed);
                nexp=2;
                if(pexp<=0.40){
                        mic[xcur][ycur][zcur]=ETTR;
                        nexp=1;
                }
                else{
                        mic[xcur][ycur][zcur]=DIFFC3A;
                        count[DIFFC3A]+=1;
        /* indicate that diffusing species remains in current location */
                        action=7;
                        nexp=2;
                }

    /* Perform expansion that occurs when ettringite is formed */
    /* xexp, yexp and zexp are the coordinates of the last ettringite */
    /* pixel to be added */
                xexp=xnew;
                yexp=ynew;
                zexp=znew;
                for(iexp=1;iexp<=nexp;iexp++){
                        newact=extettr(xexp,yexp,zexp);
                        /* update xexp, yexp and zexp */
                        switch (newact){
                                case 1:
                                        xexp-=1;
                                        if(xexp<0){xexp=(SYSIZE-1);}
                                        break;
                                case 2:
                                        xexp+=1;
```

```
                                                if(xexp>=SYSIZE){xexp=0;}
                                                break;
                                        case 3:
                                                yexp-=1;
                                                if(yexp<0){yexp=(SYSIZE-1);}
                                                break;
                                        case 4:
                                                yexp+=1;
                                                if(yexp>=SYSIZE){yexp=0;}
                                                break;
                                        case 5:
                                                zexp-=1;
                                                if(zexp<0){zexp=(SYSIZE-1);}
                                                break;
                                        case 6:
                                                zexp+=1;
                                                if(zexp>=SYSIZE){zexp=0;}
                                                break;
                                        default:
                                                break;
                                }
                        }

                /* probabilistic-based expansion for last ettringite pixel */
                        pexp=ran1(seed);
                        if(pexp<=0.30){
                                newact=extettr(xexp,yexp,zexp);
                        }
                }
/* check for reaction with diffusing or solid ettringite to form AFm */
/* reaction at solid ettringite only possible if ettringite is soluble */
/* and even then on a limited bases to avoid a great formation of AFm */
/* when ettringite first becomes soluble */
                pgrow=ran1(seed);
   if((check==DIFFETTR)||((check==ETTR)&&(soluble[ETTR]==1)&&(pgrow<=C3AETTR))){
                /* convert diffusing or solid ettringite to AFm */
                mic[xnew][ynew][znew]=AFM;
                /* decrement counts of diffusing C3A and ettringite */
                count[DIFFC3A]-=1;
                if(check==DIFFETTR){
                        count[DIFFETTR]-=1;
                }
                action=0;

                /* convert diffusing C3A to AFm or leave at diffusing C3A */
                pexp=ran1(seed);
                if(pexp<=0.2424){
```

```
                        mic[xcur][ycur][zcur]=AFM;
                        pafm=(-0.1);
                }
                else{
                        mic[xcur][ycur][zcur]=DIFFC3A;
                        count[DIFFC3A]+=1;
                        action=7;
                        pafm=(0.278-0.2424)/(1.-0.2424);
                }

                /* probabilistic-based expansion for new AFm pixel */
                pexp=ran1(seed);
                if(pexp<=pafm){
                        extafm(xnew,ynew,znew);
                }
        }
        if((action!=0)&&(action!=7)){

                /* if diffusion is possible, execute it */
                if(check==POROSITY){
                        mic[xcur][ycur][zcur]=POROSITY;
                        mic[xnew][ynew][znew]=DIFFC3A;
                }
                else{
                        /* indicate that diffusing C3A remained */
                        /* at original location */
                        action=7;
                }
        }
        }
        return(action);
}

/* routine to oversee hydration by updating position of all */
/* remaining diffusing species */
/* Calls movech, movec3a, movefh3, moveettr, movecsh, and movegyp */
void hydrate (fincyc,stepmax,chpar1,chpar2,hgpar1,hgpar2,fhpar1,fhpar2)
        int fincyc,stepmax;
        float chpar1,chpar2,hgpar1,hgpar2,fhpar1,fhpar2;
{
        int xpl,ypl,zpl,phpl,xpnew,ypnew,zpnew;
        float chprob,c3ah6prob,fh3prob;
        long int icnt,nleft,ntodo,ndale;
        int istep,termflag,reactf;
        float beterm;
        struct ants *curant,*antgone;
```

```
        ntodo=nmade;
        nleft=nmade;
        termflag=0;

/* Perform diffusion until all reacted or max. # of diffusion steps reached */
        for(istep=1;((istep<=stepmax)&&(nleft>0));istep++){
                if((fincyc==1)&&(istep==stepmax)){termflag=1;}

                nleft=0;
                ndale=0;

                /* determine probabilities for CH and C3AH6 nucleation */
                beterm=exp(-(double)(count[DIFFCH])/chpar2);
                chprob=chpar1*(1.-beterm);
                beterm=exp(-(double)(count[DIFFC3A])/hgpar2);
                c3ah6prob=hgpar1*(1.-beterm);
                beterm=exp(-(double)(count[DIFFFH3])/fhpar2);
                fh3prob=fhpar1*(1.-beterm);

                /* Process each diffusing species in turn */
                curant=headant->nextant;
                while(curant!=NULL){
                        ndale+=1;
                        xpl=curant->x;
                        ypl=curant->y;
                        zpl=curant->z;
                        phpl=curant->id;

    /* based on ID, call appropriate routine to process diffusing species */
                        switch (phpl) {
                                case DIFFCSH:
                                        reactf=movecsh(xpl,ypl,zpl,termflag);
                                        break;
                                case DIFFCH:
                                  reactf=movech(xpl,ypl,zpl,termflag,chprob);
                                  break;
                                case DIFFFH3:
                                  reactf=movefh3(xpl,ypl,zpl,termflag,fh3prob);
                                  break;
                                case DIFFGYP:
                                        reactf=movegyp(xpl,ypl,zpl,termflag);
                                        break;
                                case DIFFC3A:
                                 reactf=movec3a(xpl,ypl,zpl,termflag,c3ah6prob);
                                 break;
                                case DIFFETTR:
                                        reactf=moveettr(xpl,ypl,zpl,termflag);
```

```
                break;
        default:
                printf("Error in ID of phase \n");
                break;
}

/* if no reaction */
if(reactf!=0){
        nleft+=1;
        xpnew=xpl;
        ypnew=ypl;
        zpnew=zpl;

        /* update location of diffusing species */
        switch (reactf) {
                case 1:
                        xpnew-=1;
                        if(xpnew<0){xpnew=(SYSIZE-1);}
                        break;
                case 2:
                        xpnew+=1;
                        if(xpnew>=SYSIZE){xpnew=0;}
                        break;
                case 3:
                        ypnew-=1;
                        if(ypnew<0){ypnew=(SYSIZE-1);}
                        break;
                case 4:
                        ypnew+=1;
                        if(ypnew>=SYSIZE){ypnew=0;}
                        break;
                case 5:
                        zpnew-=1;
                        if(zpnew<0){zpnew=(SYSIZE-1);}
                        break;
                case 6:
                        zpnew+=1;
                        if(zpnew>=SYSIZE){zpnew=0;}
                        break;
                default:
                        break;
        }

        /* store new location of diffusing species */
        curant->x=xpnew;
        curant->y=ypnew;
        curant->z=zpnew;
```

```
                                        curant->id=phpl;
                                        curant=curant->nextant;
                                } /* end of reactf!=0 loop */
                                /* else remove ant from list */
                                else{
                                        if(ndale==1){
                                                headant->nextant=curant->nextant;
                                        }
                                        else{
                                             (curant->prevant)->nextant=curant->nextant;
                                        }
                                        if(curant->nextant!=NULL){
                                             (curant->nextant)->prevant=curant->prevant;
                                        }
                                        else{
                                                tailant=curant->prevant;
                                        }
                                        antgone=curant;
                                        curant=curant->nextant;
                                        free(antgone);
                                        ngoing-=1;
                                }
                        } /* end of curant loop */
                        ntodo=nleft;
                } /* end of istep loop */
}
```

C.5 Listing for disreal3d.c

```
/****************************************************************************/
/*                                                                          */
/*      Program disreal3d.c to hydrate three-dimensional cement             */
/*              particles in a 3-D box with periodic boundaries.            */
/*              Use cellular automata techniques and preserves correct      */
/*              hydration volume stoichiometry.                             */
/*      Programmer: Dale P. Bentz                                           */
/*                      Building and Fire Research Laboratory               */
/*                      NIST                                                */
/*                      Building 226 Room B-350                             */
/*                      Gaithersburg, MD  20899   USA                       */
/*                      (301) 975-5865       FAX: 301-990-6891              */
/*                      E-mail: dale.bentz@nist.gov                         */
/*                                                                          */
/****************************************************************************/
/* Three-dimensional version created 7/94 */
/* General C version created 8/94 */
```

```
/* Modified to have pseudo-continuous dissolution 11/94 */
/* Difference between disreal3.c and this program, disreal3d.c, */
/* is that in this program, dissolved silicates are located close */
/* to dissolution source within a 17*17*17 box centered at dissolution source */
/* Heat of formation data added: 12/92 */
/* Hydration under self-desiccating conditions included 9/95 */
/* Additions for adiabatic hydration conditions included 9/96 */

#include <stdio.h>
#include <math.h>

#define CUBEMAX 7        /* Maximum cube size for checking pore size */
#define CUBEMIN 3        /* Minimum cube size for checking pore size */
#define SYSIZE 100      /* System size in pixels */
#define DISBIAS 20.0    /* Dissolution bias- to change all dissolution rates */
                        /* All dissolution rates are divided by this value */
                        /* To examine early setting, value should be */
                        /* increased to perhaps 100 or so */
#define DETTRMAX 1200   /* Maximum allowed # of ettringite diffusing species */
#define CHCRIT 50.0     /* Scale parameter to adjust CH dissolution probability */
                        /* based on number of CH diffusing species */
#define C3AH6CRIT 10.0     /* Scale par. to adjust C3AH6 dissolution prob. */
                           /* based on potential sulfate (gypsum) in solution */
#define C3AH6GROW 0.01     /* Probability for C3AH6 growth */
#define ETTRGROW 0.002     /* Probability for ettringite growth */
#define C3AETTR 0.001      /* Probability for reaction of diffusing C3A
                              with ettringite  */
/* define IDs for each phase used in model */
#define POROSITY 0
#define C3S 1
#define C2S 2
#define CH 3
#define CSH 4
#define C3A 5
#define GYPSUM 6
#define C4AF 7
#define C3AH6 8
#define ETTR 9
#define AFM 10
#define FH3 11
#define POZZ 12
#define INERT 13
#define INERTAGG 14
#define ABSGYP 15
#define DIFFCSH 16
#define DIFFCH 17
#define DIFFGYP 18
```

```
#define DIFFC3A 19
#define DIFFFH3 20
#define DIFFETTR 21
#define EMPTYP 22          /* Empty porosity due to self desiccation */
#define OFFSET 30          /* Offset for highlighted potentially soluble pixel */
/* define heat capacities for all components in J/g/C */
/* including free and bound water */
#define Cp_cement 0.75
#define Cp_pozz 0.75
#define Cp_agg 0.84
#define Cp_h2o 4.18      /* Cp for free water */
#define Cp_bh2o 2.2      /* Cp for bound water */
#define WN 0.23         /* water bound per gram of cement during hydration */
#define WCHSH 0.06      /* water imbibed per gram of cement during chemical
 shrinkage (estimate) */

/* data structure for diffusing species - to be dynamically allocated */
/* Use of a doubly linked list to allow for easy maintenance */
/* (i.e. insertion and deletion) */
/* Added 11/94 */
struct ants{
        char x,y,z,id;
        struct ants *nextant;
        struct ants *prevant;
};
/* data structure for elements to remove to simulate self-desiccation */
/* once again a doubly linked list */
struct togo{
        int x,y,z,npore;
        struct togo *nexttogo;
        struct togo *prevtogo;
};

/* Global variables */
/* Microstructure stored in array mic of type char to minimize storage */
/* Initial particle IDs stored in array micpart (for assessing set point) */
static char mic [SYSIZE] [SYSIZE] [SYSIZE];
static short int micpart [SYSIZE] [SYSIZE] [SYSIZE];
/* counts for dissolved and solid species */
long int discount[EMPTYP+1], count[EMPTYP+1];
/* Counts for pozzolan reacted, initial pozzolan, gypsum, ettringite,
and initial porosity */
long int npr,nfill,ncsbar,netbar,porinit;
/* Initial clinker phase counts */
long int c3sinit,c2sinit,c3ainit,c4afinit;
long int nmade,ngoing,gypready,poregone,poretodo;
int cyccnt,cubesize,sealed;
```

```
/* Parameters for kinetic modelling ---- maturity approach */
float ind_time,temp_0,temp_cur,time_cur,E_act,beta,heat_cf,w_to_c,s_to_c;
float alpha_cur,heat_old,heat_new,cemmass,mass_agg,mass_water,mass_fill,Cp_now;
/* Arrays for dissolution probabilities for each phase */
float disprob[EMPTYP+1],disbase[EMPTYP+1],gypabsprob;
/* Arrays for specific gravities, molar volumes, heats of formation, and */
/* molar water consumption for each phase */
float specgrav[EMPTYP+1],molarv[EMPTYP+1],heatf[EMPTYP+1],waterc[EMPTYP+1];
/* Solubility flags and diffusing species created for each phase */
int soluble[EMPTYP+1], creates[EMPTYP+1];
int *seed;       /* Random number seed */
struct ants *headant,*tailant;

/* Supplementary programs */
#include "ran1.c"
#include "burn3d.c"
#include "burnset.c"
#include "hydreal3d.c"

/* routine to initialize values for solubilities, molar volumes, etc. */
/* Called by main program */
/* Calls no other routines */
void init()
{
        int i;

        for(i=0;i<=EMPTYP;i++){
                creates[i]=0;
                soluble[i]=0;
                disprob[i]=0.0;
                disbase[i]=0.0;
        }

        /* soluble [x] - flag indicating if phase x is soluble */
        /* disprob [x] - probability of dissolution (relative diss. rate) */
        gypabsprob=0.01; /* One sulfate absorbed per 100 CSH units */
        /* Note that for the first cycle, only the aluminates */
        /* and gypsum are soluble */
        soluble[C4AF]=1;
        disprob[C4AF]=0.1/DISBIAS;
        disbase[C4AF]=0.1/DISBIAS;
        creates[C4AF]=POROSITY;
        soluble[C3S]=0;
        disprob[C3S]=0.95/DISBIAS;
        disbase[C3S]=0.95/DISBIAS;
        creates[C3S]=DIFFCSH;
        soluble[C2S]=0;
```

```
disprob[C2S]=0.15/DISBIAS;
disbase[C2S]=0.15/DISBIAS;
creates[C2S]=DIFFCSH;
soluble[C3A]=1;
disprob[C3A]=0.8/DISBIAS;
disbase[C3A]=0.8/DISBIAS;
creates[C3A]=POROSITY;
soluble[GYPSUM]=1;
disprob[GYPSUM]=0.1;
disbase[GYPSUM]=0.1;
creates[GYPSUM]=POROSITY;
soluble[CH]=0;
disprob[CH]=0.1/DISBIAS;
disbase[CH]=0.1/DISBIAS;
creates[CH]=DIFFCH;
soluble[C3AH6]=1;
disprob[C3AH6]=0.5/DISBIAS;
disbase[C3AH6]=0.5/DISBIAS;
creates[C3AH6]=POROSITY;
/* Ettringite is initially insoluble */
soluble[ETTR]=0;
disbase[ETTR]=0.10/DISBIAS;
disprob[ETTR]=0.10/DISBIAS;
creates[ETTR]=DIFFETTR;

/* establish molar volumes and heats of formation */
/* molar volumes are in cm^3/mole */
/* heats of formation are in kJ/mole */
/* See paper by Fukuhara et al., Cem. & Conc. Res., 11, 407-14, 1981. */
/* values for Porosity are those of water */
molarv[POROSITY]=18.0;
heatf[POROSITY]=(-285.83);
waterc[POROSITY]=1.0;
specgrav[POROSITY]=1.0;

molarv[C3S]=71.0;
heatf[C3S]=(-2927.82);
waterc[C3S]=0.0;
specgrav[C3S]=3.21;
/* For improvement in chemical shrinkage correspondence */
/* Changed molar volume of C-S-H to 108 5/24/95 */
molarv[CSH]=108.;
heatf[CSH]=(-3283.0);
waterc[CSH]=4.0;
specgrav[CSH]=2.12;

molarv[CH]=33.1;
```

```
        heatf[CH]=(-986.1);
        waterc[CH]=1.0;
        specgrav[CH]=2.24;

        molarv[C2S]=52.;
        heatf[C2S]=(-2311.6);
        waterc[C2S]=0.0;
        specgrav[C2S]=3.28;

        molarv[C3A]=89.1;
        heatf[C3A]=(-3587.8);
        waterc[C3A]=0.0;
        specgrav[C3A]=3.03;

        molarv[GYPSUM]=74.2;
        heatf[GYPSUM]=(-2022.6);
        waterc[GYPSUM]=0.0;
        specgrav[GYPSUM]=2.32;

        molarv[C4AF]=128.;
        heatf[C4AF]=(-5090.3);
        waterc[C4AF]=0.0;
        specgrav[C4AF]=3.73;

        molarv[C3AH6]=150.;
        heatf[C3AH6]=(-5548.);
        waterc[C3AH6]=6.0;
        specgrav[C3AH6]=2.52;

/* Changed molar volume of FH3 to 69.8 (specific gravity of 3.0) 5/23/95 */
        molarv[FH3]=69.8;
        heatf[FH3]=(-823.9);
        waterc[FH3]=3.0;
        specgrav[FH3]=3.0;

/* Changed molar value of ettringite to 735 (spec. gr.=1.7)  5/24/95 */
        molarv[ETTR]=735;
        heatf[ETTR]=(-17539.0);
        /* Each mole of ETTR which forms requires 32 moles of water, */
        /* six of which are supplied by 3 moles of gypsum in forming ETTR */
        /* leaving 26 moles to be incorporated from free water */
        waterc[ETTR]=26.0;
        specgrav[ETTR]=1.7;

        molarv[AFM]=313;
        heatf[AFM]=(-8778.0);
        /* Each mole of AFM which forms requires 12 moles of water, */
```

```
        /* two of which are supplied by gypsum in forming ETTR  */
        /* leaving 10 moles to be incorporated from free water */
        waterc[AFM]=10.0;
        specgrav[AFM]=1.99;

        molarv[POZZ]=27.0;
        /* Use heat of formation of SiO2 (quartz) for unreacted pozzolan */
        /* Source- Handbook of Chemistry and Physics */
        heatf[POZZ]=-907.5;
        waterc[POZZ]=0.0;
        specgrav[POZZ]=2.2;

/* Assume inert filler has same specific gravity and molar volume as SiO2 */
        molarv[INERT]=27.0;
        heatf[INERT]=0.0;
        waterc[INERT]=0.0;
        specgrav[INERT]=2.2;

        molarv[ABSGYP]=74.2;
        heatf[ABSGYP]=(-2022.6);
        waterc[ABSGYP]=0.0;
        specgrav[ABSGYP]=2.32;

        molarv[EMPTYP]=18.0;
        heatf[EMPTYP]=(-285.83);
        waterc[EMPTYP]=0.0;
        specgrav[EMPTYP]=1.0;
}

/* routine to check if a pixel located at (xck.yck,zck) is on an edge
/* (in contact with pore space) in 3-D system */
/* Called by passone */
/* Calls no other routines */
int chckedge(xck,yck,zck)
        int xck,yck,zck;
{
        int edgeback,x2,y2,z2;
        int ip;

        edgeback=0;

        /* Check all six neighboring pixels */
        for(ip=1;((ip<=6)&&(edgeback==0));ip++){

                x2=xck;
                y2=yck;
                z2=zck;
```

```
		switch (ip){
			case 1:
				x2+=1;
				if(x2>=SYSIZE){x2=0;}
				break;
			case 2:
				x2-=1;
				if(x2<0){x2=SYSIZE-1;}
				break;
			case 3:
				y2+=1;
				if(y2>=SYSIZE){y2=0;}
				break;
			case 4:
				y2-=1;
				if(y2<0){y2=SYSIZE-1;}
				break;
			case 5:
				z2+=1;
				if(z2>=SYSIZE){z2=0;}
				break;
			case 6:
				z2-=1;
				if(z2<0){z2=SYSIZE-1;}
				break;
			default:
				break;
		}
		if(mic [x2] [y2] [z2]==POROSITY){
			edgeback=1;
		}
	}
	return(edgeback);
}

/* routine for first pass through microstructure during dissolution */
/* low and high indicate phase ID range to check for surface sites */
/* Called by dissolve */
/* Calls chckedge */
void passone(low,high,cycid)
	int low,high,cycid;
{
	int i,xid,yid,zid,phid,iph,edgef;

	/* gypready used to determine if any soluble gypsum remains */
	if((low<=GYPSUM)&&(GYPSUM<=high)){
		gypready=0;
```

```
	}
	/* Scan the entire 3-D microstructure */
	for(xid=0;xid<SYSIZE;xid++){
	for(yid=0;yid<SYSIZE;yid++){
	for(zid=0;zid<SYSIZE;zid++){

	/* Identify phase and update count */
	phid=50;
	for(i=low;((i<=high)&&(phid==50));i++){

		if(mic [xid][yid][zid]==i){
			phid=i;
			/* Update count for this phase */
			count[i]+=1;
			if((low<=GYPSUM)&&(GYPSUM<=high)&&(i==GYPSUM)){
				gypready+=1;
			}
			/* If first cycle, then accumulate initial counts */
			if(cycid==1){
				if(i==POROSITY){porinit+=1;}
				else if(i==C3S){c3sinit+=1;}
				else if(i==C2S){c2sinit+=1;}
				else if(i==C3A){c3ainit+=1;}
				else if(i==C4AF){c4afinit+=1;}
				else if(i==GYPSUM){ncsbar+=1;}
				else if(i==POZZ){nfill+=1;}
				else if(i==ETTR){netbar+=1;}
			}
		}
	}

	if(phid!=50){
		/* If phase is soluble, see if it is in contact with porosity */
		if((cycid!=0)&&(soluble[phid]==1)){
			edgef=chckedge(xid,yid,zid);
			if(edgef==1){
/* Surface eligible species has an ID OFFSET greater than its original value */
				mic [xid][yid][zid]+=OFFSET;
			}
		}
	}
	}  /* end of zid */
	}  /* end of yid */
	}  /* end of xid */
}

/* routine to locate a diffusing csh species near dissolution source */
```

```
/* at (xcur,ycur,zcur) */
/* Called by dissolve */
/* Calls no other routines */
int loccsh(xcur,ycur,zcur)
        int xcur,ycur,zcur;
{
        int xpmax,ypmax,effort,tries,xmod,ymod,zmod;
        struct ants *antnew;

        effort=0;    /* effort indicate appropriate location found */
        tries=0;
        /* 500 tries in immediate vicinity */
        while((effort==0)&&(tries<500)){
                tries+=1;
                xmod=(-8)+(int)(17.*ran1(seed));
                ymod=(-8)+(int)(17.*ran1(seed));
                zmod=(-8)+(int)(17.*ran1(seed));
                if(xmod>8){xmod=8;}
                if(ymod>8){ymod=8;}
                if(zmod>8){ymod=8;}
                xmod+=xcur;
                ymod+=ycur;
                zmod+=zcur;
                /* Periodic boundaries */
                if(xmod>=SYSIZE){xmod-=SYSIZE;}
                else if(xmod<0){xmod+=SYSIZE;}
                if(zmod>=SYSIZE){zmod-=SYSIZE;}
                else if(zmod<0){zmod+=SYSIZE;}
                if(ymod<0){ymod+=SYSIZE;}
                else if(ymod>=SYSIZE){ymod-=SYSIZE;}

                if(mic[xmod][ymod][zmod]==POROSITY){
                        effort=1;
                        mic[xmod][ymod][zmod]=DIFFCSH;
                        nmade+=1;
                        ngoing+=1;
                        /* Add this diffusing species to the linked list */
                        antnew=(struct ants *)malloc(sizeof(struct ants));
                        antnew->x=xmod;
                        antnew->y=ymod;
                        antnew->z=zmod;
                        antnew->id=DIFFCSH;
                      /* Now connect this ant structure to end of linked list */
                        antnew->prevant=tailant;
                        tailant->nextant=antnew;
                        antnew->nextant=NULL;
                        tailant=antnew;
```

```
                }
        }
        return(effort);
}

/* routine to count number of pore pixels in a cube of size boxsize */
/* centered at (qx,qy,qz) */
/* Called by makeinert */
/* Calls no other routines */
int countbox(boxsize,qx,qy,qz)
        int boxsize,qx,qy,qz;
{
        int nfound,ix,iy,iz;
        int hx,hy,hz,boxhalf;

        boxhalf=boxsize/2;
        nfound=0;
        /* Count the number of requisite pixels in the 3-D cube box */
        /* using periodic boundaries */
        for(ix=(qx-boxhalf);ix<=(qx+boxhalf);ix++){
                hx=ix;
                if(hx<0){hx+=SYSIZE;}
                else if(hx>=SYSIZE){hx-=SYSIZE;}
        for(iy=(qy-boxhalf);iy<=(qy+boxhalf);iy++){
                hy=iy;
                if(hy<0){hy+=SYSIZE;}
                else if(hy>=SYSIZE){hy-=SYSIZE;}
        for(iz=(qz-boxhalf);iz<=(qz+boxhalf);iz++){
                hz=iz;
                if(hz<0){hz+=SYSIZE;}
                else if(hz>=SYSIZE){hz-=SYSIZE;}
                /* Count if porosity, diffusing species, or empty porosity */
                if((mic [hx] [hy] [hz]<C3S)||(mic[hx] [hy] [hz]>ABSGYP)){
                        nfound+=1;
                }
        }
        }
        }
        return(nfound);
}

/* routine to create ndesire pixels of empty pore space to simulate */
/* self-desiccation */
/* Called by dissolve */
/* Calls countbox */
void makeinert(ndesire)
        long int ndesire;
```

```
{
        struct togo *headtogo,*tailtogo,*newtogo,*lasttogo,*onetogo;
        long int idesire;
        int px,py,pz,placed,cntpore,cntmax;

        /* First allocate the linked list */
        headtogo=(struct togo *)malloc(sizeof(struct togo));
        headtogo->x=headtogo->y=headtogo->z=(-1);
        headtogo->npore=0;
        headtogo->nexttogo=NULL;
        headtogo->prevtogo=NULL;
        tailtogo=headtogo;
        cntmax=0;

        printf("In makeinert with %ld needed elements \n",ndesire);
        fflush(stdout);
        /* Add needed number of elements to the end of the list */
        for(idesire=2;idesire<=ndesire;idesire++){
                newtogo=(struct togo *)malloc(sizeof(struct togo));
                newtogo->npore=0;
                newtogo->x=newtogo->y=newtogo->z=(-1);
                tailtogo->nexttogo=newtogo;
                newtogo->prevtogo=tailtogo;
                tailtogo=newtogo;
        }

        /* Now scan the microstructure and rank the sites */
        for(px=0;px<SYSIZE;px++){
        for(py=0;py<SYSIZE;py++){
        for(pz=0;pz<SYSIZE;pz++){
                if(mic[px][py][pz]==POROSITY){
                        cntpore=countbox(cubesize,px,py,pz);
                        if(cntpore>cntmax){cntmax=cntpore;}
                        /* Store this site value at appropriate place in */
                        /* sorted linked list */
                        if(cntpore>(tailtogo->npore)){
                                placed=0;
                                lasttogo=tailtogo;
                                while(placed==0){
                                        newtogo=lasttogo->prevtogo;
                                        if(newtogo==NULL){
                                                placed=2;
                                        }
                                        else{
                                                if(cntpore<=(newtogo->npore)){
                                                        placed=1;
                                                }
```

```
                                        }
                                        if(placed==0){
                                                lasttogo=newtogo;
                                        }
                                }
                        onetogo=(struct togo *)malloc(sizeof(struct togo));
                                onetogo->x=px;
                                onetogo->y=py;
                                onetogo->z=pz;
                                onetogo->npore=cntpore;
                                /* Insertion at the head of the list */
                                if(placed==2){
                                        onetogo->prevtogo=NULL;
                                        onetogo->nexttogo=headtogo;
                                        headtogo->prevtogo=onetogo;
                                        headtogo=onetogo;
                                }
                                if(placed==1){
                                        onetogo->nexttogo=lasttogo;
                                        onetogo->prevtogo=newtogo;
                                        lasttogo->prevtogo=onetogo;
                                        newtogo->nexttogo=onetogo;
                                }
                                /* Eliminate the last element */
                                lasttogo=tailtogo;
                                tailtogo=tailtogo->prevtogo;
                                tailtogo->nexttogo=NULL;
                                free(lasttogo);
                }
        }
}
}
}

/* Now remove the sites */
/* starting at the head of the list */
/* and deallocate all of the used memory */
for(idesire=1;idesire<=ndesire;idesire++){
        px=headtogo->x;
        py=headtogo->y;
        pz=headtogo->z;
        if(px!=(-1)){
                mic [px] [py] [pz]=EMPTYP;
                count[POROSITY]-=1;
                count[EMPTYP]+=1;
        }
        lasttogo=headtogo;
```

```
                headtogo=headtogo->nexttogo;
                free(lasttogo);
        }
        /* If only small cubes of porosity were found, then adjust */
        /* cubesize to have a more efficient search in the future */
        if(cubesize>CUBEMIN){
                if((2*cntmax)<(cubesize*cubesize*cubesize)){
                        cubesize-=2;
                }
        }
}


/* routine to implement a cycle of dissolution */
/* Called by main program */
/* Calls passone, loccsh, and makeinert */
void dissolve(cycle)
        int cycle;
{
        int nc3aext,ncshext,nchext,nfh3ext,ngypext,plok,edgef;
        int xpmax,ypmax,phid,phnew,plnew,cread;
        int i,xloop,yloop,zloop,ngood,ix1,iy1,xc,yc,valid,xc1,yc1;
        int iz1,zc,zc1;
        long int water_left,ctest;
        int placed,cshrand,ntrycsh,maxsulfate;
        long int nsurf,suminit;
        long int xext,nhgd;
        float pdis,plfh3,fchext,fc3aext,mass_now,tot_mass,heatfill;
        float heatsum,molesh2o,molesdh2o,h2oinit,alpha,heat2,heat3,heat4;
        FILE *heatfile,*phfile;
        struct ants *antadd;

        /* Initialize variables */
        nmade=0;
        cshrand=0;      /* counter for number of csh diffusing species to */
                        /* be located at random locations in microstructure */
        heat_old=heat_new; /* new and old values for heat released */

        /* Initialize dissolution and phase counters */
        nsurf=0;
        for(i=0;i<=EMPTYP;i++){
                discount[i]=0;
                count[i]=0;
        }

        /* Pass one- highlight all edge points which are soluble */
        soluble[C3AH6]=0;
        soluble[CH]=0;
```

```
        passone(0,EMPTYP,cycle);

        /* If first cycle, then determine all mixture proportions based */
        /* on user input and original microstructure */
        if(cycle==1){
                /* Mass of cement in system */
                cemmass=(specgrav[C3S]*(float)count[C3S]+specgrav[C2S]*
                (float)count[C2S]+specgrav[C3A]*(float)count[C3A]+
                specgrav[C4AF]*(float)count[C4AF]+specgrav[GYPSUM]*
                (float)count[GYPSUM]);
                /* Total mass in system neglecting single aggregate */
                tot_mass=cemmass+(float)count[POROSITY]+specgrav[INERT]*
                (float)count[INERT]+specgrav[POZZ]*(float)count[POZZ];
                /* water-to-cement ratio */
                w_to_c=(float)count[POROSITY]/cemmass;
                mass_water=(1.-mass_agg)*(float)count[POROSITY]/tot_mass;
                /* pozzolan-to-cement ratio */
                s_to_c=(float)(count[INERT]*specgrav[INERT]+
                count[POZZ]*specgrav[POZZ])/cemmass;
                /* Conversion factor to kJ/kg for heat produced */
                heatfill=(float)(count[INERT]+count[POZZ])/cemmass;
                heat_cf=0.001*(0.3125+w_to_c+heatfill);
                mass_fill=(1.-mass_agg)*(float)(count[INERT]*specgrav[INERT]+
                count[POZZ]*specgrav[POZZ])/tot_mass;
                printf("Calculated w/c is %.4f\n",w_to_c);
                printf("Calculated s/c is %.4f \n",s_to_c);
                printf("Calculated heat conversion factor is %f \n",heat_cf);
printf("Calculated mass fractions of water and filler are %.4f  and %.4f \n",
                mass_water,mass_fill);
        }

        heatsum=molesh2o=molesdh2o=0.0;
        alpha=0.0;      /* degree of hydration */
        /* heat2 is heat of hydration based on heats of hydration of pure */
        /* compounds (c3s,c2s,c3a,c4af) from Neville and Brooks */
        /* page 14, Table 2.4 */
        /* Originally from Lerch and Bogue,  */
        /* See book (The Chemistry of Cement) by Lea for complete ref. */
        /* heat3 is heat of hydration based on heats of hydration of pure */
        /* compounds (c3s,c2s,c3a,c4af) from Cement Chemistry by Taylor */
        /* page 232, Table 7.5 */
        /* heat4 contains measured heat release for C4AF hydration from  */
        /* Fukuhara et al., Cem. and Conc. Res. article */
        heat2=heat3=heat4=0.0;
        mass_now=0.0;    /* total cement mass corrected for hydration */
        suminit=c3sinit+c2sinit+c3ainit+c4afinit+ncsbar;
        /* ctest is number of gypsum likely to form ettringite */
```

```
/* 1 unit of C3A can react with 2.5 units of Gypsum */
ctest=count[DIFFGYP];
if((float)ctest>(2.5*(float)count[DIFFC3A])){
          ctest=2.5*(float)count[DIFFC3A];
}
for(i=0;i<=EMPTYP;i++){
        if((i!=0)&&(i<=ABSGYP)&&(i!=INERTAGG)){
                heatsum+=(float)count[i]*heatf[i]/molarv[i];
        /* Tabulate moles of H2O consumed by reactions so far */
                molesh2o+=(float)count[i]*waterc[i]/molarv[i];
        }
        /* assume that all C3A which can, does form ettringite */
        if(i==DIFFC3A){
   heatsum+=((float)count[i]-(float)ctest/2.5)*heatf[C3A]/molarv[C3A];
        }
        /* assume all gypsum which can, does form ettringite */
        /* rest will remain as gypsum */
        if(i==DIFFGYP){
         heatsum+=(float)(count[i]-ctest)*heatf[GYPSUM]/molarv[GYPSUM];
                heatsum+=(float)ctest*3.30*heatf[ETTR]/molarv[ETTR];
                molesdh2o+=(float)ctest*3.30*waterc[ETTR]/molarv[ETTR];
        }
        else if(i==DIFFCH){
                heatsum+=(float)count[i]*heatf[CH]/molarv[CH];
                molesdh2o+=(float)count[i]*waterc[CH]/molarv[CH];
        }
        else if(i==DIFFCSH){
                heatsum+=(float)count[i]*heatf[CSH]/molarv[CSH];
                molesdh2o+=(float)count[i]*waterc[CSH]/molarv[CSH];
        }
        else if(i==DIFFETTR){
                heatsum+=(float)count[i]*heatf[ETTR]/molarv[ETTR];
                molesdh2o+=(float)count[i]*waterc[ETTR]/molarv[ETTR];
        }
        else if(i==C3S){
                alpha+=(float)(c3sinit-count[i]);
                mass_now+=specgrav[C3S]*(float)count[C3S];
                heat2+=.502*(float)(c3sinit-count[i])*specgrav[C3S];
                heat3+=.517*(float)(c3sinit-count[i])*specgrav[C3S];
                heat4+=.517*(float)(c3sinit-count[i])*specgrav[C3S];
        }
        else if(i==C2S){
                alpha+=(float)(c2sinit-count[i]);
                mass_now+=specgrav[C2S]*(float)count[C2S];
                heat2+=.260*(float)(c2sinit-count[i])*specgrav[C2S];
                heat3+=.262*(float)(c2sinit-count[i])*specgrav[C2S];
                heat4+=.262*(float)(c2sinit-count[i])*specgrav[C2S];
```

```
                }
                else if(i==C3A){
                        alpha+=(float)(c3ainit-count[i]);
                        mass_now+=specgrav[C3A]*(float)count[C3A];
                        heat2+=.867*(float)(c3ainit-count[i])*specgrav[C3A];
                        heat3+=1.144*(float)(c3ainit-count[i])*specgrav[C3A];
                        heat4+=1.144*(float)(c3ainit-count[i])*specgrav[C3A];
                }
                else if(i==C4AF){
                        alpha+=(float)(c4afinit-count[i]);
                        mass_now+=specgrav[C4AF]*(float)count[C4AF];
                        heat2+=.419*(float)(c4afinit-count[i])*specgrav[C4AF];
                        heat3+=.418*(float)(c4afinit-count[i])*specgrav[C4AF];
                        heat4+=.725*(float)(c4afinit-count[i])*specgrav[C4AF];
                }
                else if(i==GYPSUM){
                        alpha+=(float)(ncsbar-count[i]);
                        mass_now+=specgrav[GYPSUM]*(float)count[GYPSUM];
                }
        }
        alpha=alpha/(float)suminit;
        /* Current degree of hydration on a mass basis */
        alpha_cur=1.0-(mass_now/cemmass);
        h2oinit=(float)porinit/molarv[POROSITY];
        /* Update water consumed to reflect that used in pozzolanic reaction */
        /* 2.1 moles of water per mole of pozzolanic C-S-H */
        /* assuming that molar volume of pozzolanic C-S-H is 80.87 */
        molesh2o+=2.1*((float)(count[POZZ]-nfill)+(float)(npr)/1.35)/80.87;
        /* Update heatsum for pozzolanic conversion */
        heatsum+=((float)count[POZZ]-(float)nfill+(float)npr/1.35)*
        (heatf[CSH]/80.87-heatf[POZZ]/molarv[POZZ]);
/* Assume 780 J/g S for pozzolanic reaction */
heat2+=0.78*((float)npr/1.35)*specgrav[POZZ];
heat3+=0.78*((float)npr/1.35)*specgrav[POZZ];
heat4+=0.78*((float)npr/1.35)*specgrav[POZZ];
        /* Adjust heat sum for water left in system */
        water_left=(long int)((h2oinit-molesh2o)*molarv[POROSITY]+0.5);
        heatsum+=(h2oinit-molesh2o-molesdh2o)*heatf[POROSITY];
        heatfile=fopen("heat.out","a");
        if(cyccnt==0){
  fprintf(heatfile,"Cycle  alpha_vol  alpha_mass  heat1 heat2  heat3  heat4\n");
        }
        fprintf(heatfile,"%d %f %f %f %f %f %f\n",
         cyccnt,alpha,alpha_cur,heatsum,heat2,heat3,heat4);
        heat_new=heat4;     /* use heat4 for all adiabatic calculations */
                            /* due to best agreement with calorimetry data */
        fclose(heatfile);
```

```
if(molesh2o>h2oinit){
        printf("All water consumed at cycle %d \n",cyccnt);
        fflush(stdout);
        exit();
}
/* Attempt to create empty porosity for account for self-desiccation */
if((sealed==1)&&((count[POROSITY]-water_left)>0)){
        poretodo=count[POROSITY]-water_left;
        makeinert(poretodo);
        poregone+=poretodo;
}
/* Output phase counts */
phfile=fopen("phases.out","a");
fprintf(phfile,"%d ",cyccnt);
for(i=0;i<=EMPTYP;i++){
        fprintf(phfile,"%ld ",count[i]);
        printf("%ld ",count[i]);
}
printf("\n");
fprintf(phfile,"%ld\n",water_left);
fclose(phfile);
cyccnt+=1;

if(cycle==0){
        return;
}
/* See if ettringite is soluble yet */
if(ncsbar!=0.0){
if((soluble[ETTR]==0)&&((count[AFM]!=0)||(((float)count[GYPSUM]/
((float)ncsbar+((float)netbar/3.21)))<0.10))){
        soluble[ETTR]=1;
        printf("Ettringite is soluble beginning at cycle %d \n",cycle);
        /* identify all new soluble ettringite */
        passone(ETTR,ETTR,2);
}
} /* end of soluble ettringite test */

/* Adjust ettringite solubility */
if(count[DIFFETTR]>DETTRMAX){
        disprob[ETTR]=0.0;
}
else{
        disprob[ETTR]=disbase[ETTR];
}
/* Adjust solubility of CH */
/* based on amount of CH currently diffusing */
/* Note that CH is always soluble to allow some */
```

```
        /* Ostwald ripening of the CH crystals */
        soluble[CH]=1;
        passone(CH,CH,2);
        if((float)count[DIFFCH]>=CHCRIT){
                disprob[CH]=disbase[CH]*CHCRIT/(float)count[DIFFCH];
        }
        else{
                disprob[CH]=disbase[CH];
        }
        /* Address solubility of C3AH6 */
        if((count[GYPSUM]>(int)((float)ncsbar*0.05))||(count[ETTR]>500)){
                soluble[C3AH6]=1;
                passone(C3AH6,C3AH6,2);
                /* Base C3AH6 solubility on maximum sulfate in solution */
                /* from gypsum or ettringite available for dissolution */
                /* The more the sulfate, the higher this solubility should be */
                maxsulfate=count[DIFFGYP];
                if((maxsulfate<count[DIFFETTR])&&(soluble[ETTR]==1)){
                        maxsulfate=count[DIFFETTR];
                }
/* Adjust C3AH6 solubility based on potential gypsum which will dissolve */
                if(maxsulfate<(int)((float)gypready*disprob[GYPSUM]*
                (float)count[POROSITY]/(float)(SYSIZE*SYSIZE*SYSIZE))){
                        maxsulfate=(int)((float)gypready*disprob[GYPSUM]*
                        (float)count[POROSITY]/(float)(SYSIZE*SYSIZE*SYSIZE));
                }
                if(maxsulfate>0){
                     disprob[C3AH6]=disbase[C3AH6]*(float)maxsulfate/C3AH6CRIT;
                     if(disprob[C3AH6]>0.5){disprob[C3AH6]=0.5;}
                }
                else{
                     disprob[C3AH6]=disbase[C3AH6];
                }
        }
        else{
                soluble[C3AH6]=0;
        }

        /* See if silicates are soluble yet */
        if((soluble[C3S]==0)&&((cycle>1)||(count[ETTR]>0)||(count[AFM]>0))){
                soluble[C2S]=1;
                soluble[C3S]=1;
                /* identify all new soluble silicates */
                passone(C3S,C2S,2);
        } /* end of soluble silicate test */

        /* Pass two- perform the dissolution of species */
```

```
nhgd=0;
/* Once again, scan all pixels in microstructure */
for(xloop=0;xloop<SYSIZE;xloop++){
for(yloop=0;yloop<SYSIZE;yloop++){
for(zloop=0;zloop<SYSIZE;zloop++){
        if(mic[xloop][yloop][zloop]>OFFSET){
                phid=mic[xloop][yloop][zloop]-OFFSET;
                /* attempt a one-step random walk to dissolve */
                plnew=(int)(6.*ran1(seed));
                if((plnew<0)||(plnew>5)){ plnew=5;}
                xc=xloop;
                yc=yloop;
                zc=zloop;
                switch(plnew){
                case 0:
                        xc-=1;
                        if(xc<0){xc=(SYSIZE-1);}
                        break;
                case 1:
                        yc-=1;
                        if(yc<0){yc=(SYSIZE-1);}
                        break;
                case 2:
                        xc+=1;
                        if(xc>=SYSIZE){xc=0;}
                        break;
                case 3:
                        yc+=1;
                        if(yc>=SYSIZE){yc=0;}
                        break;
                case 4:
                        zc-=1;
                        if(zc<0){zc=(SYSIZE-1);}
                        break;
                case 5:
                        zc+=1;
                        if(zc>=SYSIZE){zc=0;}
                        break;
                default:
                        break;
                }

               /* Generate probability for dissolution */
               pdis=ran1(seed);

               if((pdis<=disprob[phid])&&(mic[xc][yc][zc]==POROSITY)){
                        discount[phid]+=1;
```

```
                cread=creates[phid];
                mic[xloop][yloop][zloop]=POROSITY;
                if(phid==C3AH6){nhgd+=1;}
                /* Special dissolution for C4AF */
                if(phid==C4AF){
                        plfh3=ran1(seed);
                        if((plfh3<0.0)||(plfh3>1.0)){
                                plfh3=1.0;
                        }
                   /* For every C4AF that dissolves, 0.5453 */
                   /* diffusing FH3 species should be created */
                        if(plfh3<=0.5453){
                                cread=DIFFFH3;
                        }
                }
                if(cread!=POROSITY){
                        nmade+=1;
                        ngoing+=1;
                        phnew=cread;
                        count[phnew]+=1;
                        mic[xc][yc][zc]=phnew;
            antadd=(struct ants *)malloc(sizeof(struct ants));
                        antadd->x=xc;
                        antadd->y=yc;
                        antadd->z=zc;
                        antadd->id=phnew;
/* Now connect this ant structure to end of linked list */
                        antadd->prevant=tailant;
                        tailant->nextant=antadd;
                        antadd->nextant=NULL;
                        tailant=antadd;
                }
                if((phid==C3S)||(phid==C2S)){
                        plfh3=ran1(seed);
                        if((phid==C2S)||(plfh3<=0.521)){
                                placed=loccsh(xc,yc,zc);
                                if(placed!=0){
                                        count[DIFFCSH]+=1;
                                }
                                else{
                                        cshrand+=1;
                                }
                        }
                }

                if(phid==C2S){
                        plfh3=ran1(seed);
```

```
                                        if(plfh3<=0.077){
                                                placed=loccsh(xc,yc,zc);
                                                if(placed!=0){
                                                        count[DIFFCSH]+=1;
                                                }
                                                else{
                                                        cshrand+=1;
                                                }
                                        }
                                }
                        }
                        else{
                                mic[xloop][yloop][zloop]-=OFFSET;
                        }

        } /* end of if edge loop */
} /* end of zloop */
} /* end of yloop */
} /* end of xloop */

/* Now add in the extra diffusing species for dissolution */
/* Expansion factors from Young and Hansen and */
/* Mindess and Young (Concrete) */
ncshext=cshrand;
if(cshrand!=0){
           printf("cshrand is %d \n",cshrand);
}
/* CH, Gypsum, and diffusing C3A are added at totally random */
/* locations as opposed to at the dissolution site */
fchext=0.61*(float)discount[C3S]+0.191*(float)discount[C2S]+
0.2584*(float)discount[C4AF];
nchext=fchext;
if(fchext>(float)nchext){
        pdis=ran1(seed);
        if((fchext-(float)nchext)>pdis){
                nchext+=1;
        }
}
fc3aext=discount[C3A]+0.5917*(float)discount[C3AH6]+
0.696*(float)discount[C4AF];
nc3aext=fc3aext;
if(fc3aext>(float)nc3aext){
        pdis=ran1(seed);
        if((fc3aext-(float)nc3aext)>pdis){
                nc3aext+=1;
        }
}
```

```
        ngypext=discount[GYPSUM];
        count[DIFFGYP]+=ngypext;
        count[DIFFCH]+=nchext;
        count[DIFFCSH]+=ncshext;
        count[DIFFC3A]+=nc3aext;

        for(xext=1;xext<=(nc3aext+ncshext+nchext+ngypext);xext++){
        plok=0;
        do{
                xc=(int)((float)SYSIZE*ran1(seed));
                yc=(int)((float)SYSIZE*ran1(seed));
                zc=(int)((float)SYSIZE*ran1(seed));
                if(xc>=SYSIZE){xc=0;}
                if(yc>=SYSIZE){yc=0;}
                if(zc>=SYSIZE){zc=0;}

                if(mic[xc][yc][zc]==POROSITY){
                        plok=1;
                        phid=DIFFCH;
                        if(xext>nchext){phid=DIFFCSH;}
                        if(xext>(nchext+ncshext)){phid=DIFFC3A;}
                        if(xext>(nchext+ncshext+nc3aext)){phid=DIFFGYP;}
                        mic[xc][yc][zc]=phid;
                        nmade+=1;
                        ngoing+=1;
                        antadd=(struct ants *)malloc(sizeof(struct ants));
                        antadd->x=xc;
                        antadd->y=yc;
                        antadd->z=zc;
                        antadd->id=phid;
                     /* Now connect this ant structure to end of linked list */
                        antadd->prevant=tailant;
                        tailant->nextant=antadd;
                        antadd->nextant=NULL;
                        tailant=antadd;
                }
        } while (plok==0);

        } /* end of xext for extra species generation */

        printf("Dissolved- %ld  %ld  %ld  %ld %ld  %ld \n",count[DIFFCSH],
        count[DIFFCH],count[DIFFGYP],count[DIFFC3A],count[DIFFFH3],
        count[DIFFETTR]);
        printf("C3AH6 dissolved- %ld with prob. of %f \n",nhgd,disprob[C3AH6]);
        fflush(stdout);
}

/* routine to add nneed one pixel elements of phase randid at random */
```

```
/* locations in microstructure */
/* Called by main program */
/* Calls no other routines */
void addrand(randid,nneed)
        int randid;
        long int nneed;
{
        int ix,iy,iz;
        long int ic;
        int success;

        /* Add number of requested phase at random pore locations in system */
        for(ic=1;ic<=nneed;ic++){
                success=0;
                while(success==0){
                        ix=(int)((float)SYSIZE*ran1(seed));
                        iy=(int)((float)SYSIZE*ran1(seed));
                        iz=(int)((float)SYSIZE*ran1(seed));
                        if(ix==SYSIZE){ix=0;}
                        if(iy==SYSIZE){iy=0;}
                        if(iz==SYSIZE){iz=0;}
                        if(mic[ix][iy][iz]==POROSITY){
                                mic[ix][iy][iz]=randid;
                                success=1;
                        }
                }
        }
}

/* Calls dissolve and addrand */
main()
{
        int ncyc,ntimes,icyc,valin;
        int cycflag,ix,iy,iz,phtodo,setflag;
        int iseed,burnfreq,setfreq,oflag,adiaflag;
        long int nadd;
        int xpl,xph,ypl,yph,fidc3s,fidc2s,fidc3a,fidc4af,fidgyp,fidagg;
        float pnucch,pscalech,pnuchg,pscalehg,pnucfh3,pscalefh3,krate;
        float mass_cement,mass_cem_now,mass_cur;
        FILE *infile,*outfile,*adiafile;
        char filei[80],fileo[80];

        ngoing=0;
        cycflag=0;
        heat_old=heat_new=0.0;
        time_cur=0.0;      /* Elapsed time according to maturity principles */
        cubesize=CUBEMAX;
```

```
poregone=poretodo=0;
/* Get random number seed */
printf("Enter random number seed \n");
scanf("%d",&iseed);
printf("%d\n",iseed);
seed=(&iseed);

printf("Dissoluton bias is set at %f \n",DISBIAS);
printf("Do you wish to output final microstructure to file (0-No 1-Yes)\n");
scanf("%d",&oflag);
printf("%d\n",oflag);
if(oflag==1){
        printf("Enter name of file to create \n");
        scanf("%s",fileo);
        printf("%s\n",fileo);
}
/* Open file and read in original cement particle microstructure */
printf("Enter name of file to read initial microstructure from \n");
scanf("%s",filei);
printf("%s\n",filei);
/* Get phase assignments for original microstructure */
/* to transform to needed ID values */
printf("Enter IDs in file for C3S, C2S, C3A, C4AF, Gypsum, Aggregate\n");
scanf("%d %d %d %d %d %d",&fidc3s,&fidc2s,&fidc3a,&fidc4af,&fidgyp,&fidagg);
printf("%d %d %d %d %d %d\n",fidc3s,fidc2s,fidc3a,fidc4af,fidgyp,fidagg);
fflush(stdout);

infile=fopen(filei,"r");

for(ix=0;ix<SYSIZE;ix++){
for(iy=0;iy<SYSIZE;iy++){
for(iz=0;iz<SYSIZE;iz++){
        fscanf(infile,"%d",&valin);
        mic[ix][iy][iz]=POROSITY;
        if(valin==fidc3s){
                mic[ix][iy][iz]=C3S;
        }
        else if(valin==fidc2s){
                mic[ix][iy][iz]=C2S;
        }
        else if(valin==fidc3a){
                mic[ix][iy][iz]=C3A;
        }
        else if(valin==fidc4af){
                mic[ix][iy][iz]=C4AF;
        }
        else if(valin==fidgyp){
```

```
			mic[ix][iy][iz]=GYPSUM;
		}
		else if(valin==fidagg){
			mic[ix][iy][iz]=INERTAGG;
		}
}
}
}
fclose(infile);
fflush(stdout);

/* Now read in particle IDs from file */
printf("Enter name of file to read particle IDs from \n");
scanf("%s",filei);
printf("%s\n",filei);
infile=fopen(filei,"r");

for(ix=0;ix<SYSIZE;ix++){
for(iy=0;iy<SYSIZE;iy++){
for(iz=0;iz<SYSIZE;iz++){
	fscanf(infile,"%d",&valin);
	micpart[ix][iy][iz]=valin;
}
}
}

fclose(infile);
fflush(stdout);

/* Initialize counters, etc. */
npr=0;
nfill=0;
ncsbar=0;
netbar=0;
porinit=0;
cyccnt=0;
setflag=0;
c3sinit=c2sinit=c3ainit=c4afinit=0;

/* Initialize structure for ants */
headant=(struct ants *)malloc(sizeof(struct ants));
headant->prevant=NULL;
headant->nextant=NULL;
headant->x=0;
headant->y=0;
headant->z=0;
headant->id=100;	/* special ID indicating first ant in list */
```

```
        tailant=headant;

      /* Allow user to iteratively add one pixel particles of various phases */
      /* Typical application would be for addition of silica fume */
        printf("Enter number of one pixel particles to add (0 to quit) \n");
        scanf("%ld",&nadd);
        printf("%ld\n",nadd);
        while(nadd>0){
                phtodo=25;
                while((phtodo<0)||(phtodo>INERT)){
                        printf("Enter phase to add \n");
                        printf(" C3S 1 \n");
                        printf(" C2S 2 \n");
                        printf(" CH 3 \n");
                        printf(" CSH 4 \n");
                        printf(" C3A 5 \n");
                        printf(" GYPSUM 6 \n");
                        printf(" C4AF 7 \n");
                        printf(" C3AH6 8 \n");
                        printf(" ETTR 9 \n");
                        printf(" AFM 10 \n");
                        printf(" FH3 11 \n");
                        printf(" POZZ 12 \n");
                        printf(" INERT 13 \n");
                        scanf("%d",&phtodo);
                        printf("%d \n",phtodo);
                }
                addrand(phtodo,nadd);
printf("Enter number of one pixel particles to add (0 to quit) \n");
                scanf("%ld",&nadd);
                printf("%ld\n",nadd);
        }
        fflush(stdout);

        init();
        printf("After init routine \n");
        printf("Enter number of cycles to execute \n");
        scanf("%d",&ncyc);
        printf("%d \n",ncyc);
  printf("Do you wish hydration under 0) saturated or 1) sealed conditions \n");
        scanf("%d",&sealed);
        printf("%d \n",sealed);
        printf("Enter max. # of diffusion steps per cycle (500) \n");
        scanf("%d",&ntimes);
        printf("%d \n",ntimes);
        printf("Enter nuc. prob. and scale factor for CH nucleation \n");
        scanf("%f %f",&pnucch,&pscalech);
```

```
        printf("%f %f \n",pnucch,pscalech);
        printf("Enter nuc. prob. and scale factor for C3AH6 nucleation \n");
        scanf("%f %f",&pnuchg,&pscalehg);
        printf("%f %f \n",pnuchg,pscalehg);
        printf("Enter nuc. prob. and scale factor for FH3 nucleation \n");
        scanf("%f %f",&pnucfh3,&pscalefh3);
        printf("%f %f \n",pnucfh3,pscalefh3);
        printf("Enter cycle frequency for checking pore space percolation \n");
        scanf("%d",&burnfreq);
        printf("%d\n",burnfreq);
   printf("Enter cycle frequency for checking percolation of solids (set) \n");
        scanf("%d",&setfreq);
        printf("%d\n",setfreq);
        /* Parameters for adiabatic temperature rise calculation */
        printf("Enter the induction time in hours \n");
        scanf("%f",&ind_time);
        printf("%f \n",ind_time);
        time_cur+=ind_time;
        printf("Enter the initial temperature in degrees Celsius \n");
        scanf("%f",&temp_0);
        printf("%f \n",temp_0);
        temp_cur=temp_0;
        printf("Enter apparent activation energy for hydration in kJ/mole \n");
        scanf("%f",&E_act);
        printf("%f \n",E_act);
        printf("Enter kinetic factor to convert cycles to time for 25 C \n");
        scanf("%f",&beta);
        printf("%f \n",beta);
        printf("Enter mass fraction of aggregate in concrete \n");
        scanf("%f",&mass_agg);
        printf("%f \n",mass_agg);
        printf("Hydration under 0) isothermal or 1) adiabatic conditions \n");
        scanf("%d",&adiaflag);
        printf("%d \n",adiaflag);
        fflush(stdout);

        adiafile=fopen("adiabatic.out","w");
fprintf(adiafile,"Time  Temperature  Alpha  Krate     Cp_now     Mass_cem\n");
        for(icyc=1;icyc<=ncyc;icyc++){
                if(icyc==ncyc){cycflag=1;}
                dissolve(icyc);
printf("Number dissolved this pass- %ld total diffusing- %ld \n",nmade,ngoing);
                fflush(stdout);
                if(icyc==1){
                     printf("ncsbar is %ld   netbar is %ld \n",ncsbar,netbar);
                }
      hydrate(cycflag,ntimes,pnucch,pscalech,pnuchg,pscalehg,pnucfh3,pscalefh3);
```

```
		temp_0=temp_cur;
		/* Handle adiabatic case first */
		/* Cement + aggregate +water + filler=1;  that's all there is */
		mass_cement=1.-mass_agg-mass_fill-mass_water;
		if(adiaflag==1){
			/* determine heat capacity of current mixture, */
			/* accounting for imbibed water if necessary */
			if(sealed==1){
				Cp_now=mass_agg*Cp_agg;
				Cp_now+=Cp_pozz*mass_fill;
				Cp_now+=Cp_cement*mass_cement;
Cp_now+=(Cp_h2o*mass_water-alpha_cur*WN*mass_cement*(Cp_h2o-Cp_bh2o));
				mass_cem_now=mass_cement;
			}
		/* Else need to account for extra capillary water drawn in */
	/* Basis is WCHSH(0.06) g H2O per gram cement for chemical shrinkage */
		/* Need to adjust mass basis to account for extra imbibed H2O */
			else{
				mass_cur=1.+WCHSH*mass_cement*alpha_cur;
				Cp_now=mass_agg*Cp_agg/mass_cur;
				Cp_now+=Cp_pozz*mass_fill/mass_cur;
				Cp_now+=Cp_cement*mass_cement/mass_cur;
	Cp_now+=(Cp_h2o*mass_water-alpha_cur*WN*mass_cement*(Cp_h2o-Cp_bh2o));
	Cp_now+=(WCHSH*Cp_h2o*alpha_cur*mass_cement);
				mass_cem_now=mass_cement/mass_cur;
			}
		/* Determine rate constant based on Arrhenius expression */
		/* Recall that temp_cur is in degrees Celsius */
	 krate=exp(-(1000.*E_act/8.314)*((1./(temp_cur+273.15))-(1./298.15)));
		/* Update temperature based on heat generated and current Cp */
		temp_cur=temp_0+mass_cem_now*heat_cf*(heat_new-heat_old)/Cp_now;
		}
		/* Else isothermal conditions */
		else{
			krate=1.0;
			mass_cem_now=mass_cement;
		}
		/* Update time based on simple numerical integration */
		/* simulating maturity approach */
		/* with parabolic kinetics (Knudsen model) */
		if(cyccnt>1){time_cur+=(2.*(float)(cyccnt-1)-1.0)*beta/krate;}
		fprintf(adiafile,"%f %f %f %f %f %f\n",time_cur,temp_cur,
		 alpha_cur,krate,Cp_now,mass_cem_now);
		fflush(adiafile);

	/* Check percolation of pore space */
	if((icyc%burnfreq)==0){
```

```
                burn3d(0,1,0,0);
                burn3d(0,0,1,0);
                burn3d(0,0,0,1);
        }
        /* Check percolation of solids (set point) */
        if((icyc%setfreq)==0){
                setflag=burnset(1,0,0);
                setflag+=burnset(0,1,0);
                setflag+=burnset(0,0,1);
        }

        }
        fclose(adiafile);
        dissolve(0);

        /* Output final microstructure if desired */
        if(oflag==1){
                outfile=fopen(fileo,"w");

                for(ix=0;ix<SYSIZE;ix++){
                for(iy=0;iy<SYSIZE;iy++){
                for(iz=0;iz<SYSIZE;iz++){
                        fprintf(outfile,"%d\n",(int)mic[ix][iy][iz]);
                }
                }
                }
fclose(outfile);
        }
}
```

D Example input datafiles for program execution

D.1 Example input datafile for use with hydration model

```
-31283            random number seed
0                 flag indicating that final microstructure is not to be saved
cf1w030f.img      filename for file containing input 3-D microstructure
1 3 4 2 5 6       phase IDs assigned to C3S, C2S, C3A, C4AF, gypsum, and agg.
pf1wc030.img      filename for file containing particle ID microstructure
0                 number of one pixel particles to add
5000              number of cycles of hydration model to execute
1                 flag indicating hydration is to be under sealed conditions
500               maximum number of diffusion steps per cycle
0.01 9000.        parameters for CH nucleation
0.0002 100000.    parameters for C3AH6 nucleation
0.2 2500.         parameters for FH3 nucleation
200               frequency in cycles to check pore space percolation
5002              frequency in cycles to check total solids percolation
11.5              induction time in hours
25.0              initial temperature in degrees Celsius
45.7              activation energy in kJ/mole
0.0011            conversion factor to go from cycles to time
0.72              mass fraction of aggregates in mixture proportions
1                 flag indicating hydration is under adiabatic conditions
```

D.2 Example input datafile for assessing percolation of solids

```
-31283            random number seed
0                 flag indicating that final microstructure is not to be saved
c116w40f.img      filename for file containing input 3-D microstructure
1 3 2 4 5 6       phase IDs assigned to C3S, C2S, C3A, C4AF, gypsum, and agg.
p116wc40.img      filename for file containing particle ID microstructure
0                 number of one pixel particles to add
100               number of cycles of hydration model to execute
0                 flag indicating hydration is to be under saturated conditions
500               maximum number of diffusion steps per cycle
0.01 9000.        parameters for CH nucleation
0.0002 100000.    parameters for C3AH6 nucleation
0.2 2500.         parameters for FH3 nucleation
100               frequency in cycles to check pore space percolation
1                 frequency in cycles to check total solids percolation
11.5              induction time in hours
22.0              initial temperature in degrees Celsius
36.5              activation energy in kJ/mole
0.00134           conversion factor to go from cycles to time
0.72              mass fraction of aggregates in mixture proportions
```

```
0                    flag indicating hydration is under isothermal conditions
```

D.3 Example input datafile for assessing percolation of pore space

```
-31283               random number seed
0                    flag indicating that final microstructure is not to be saved
c116w40f.img         filename for file containing input 3-D microstructure
1 3 2 4 5 6          phase IDs assigned to C3S, C2S, C3A, C4AF, gypsum, and agg.
p116wc40.img         filename for file containing particle ID microstructure
0                    number of one pixel particles to add
500                  number of cycles of hydration model to execute
0                    flag indicating hydration is to be under saturated conditions
500                  maximum number of diffusion steps per cycle
0.01 9000.           parameters for CH nucleation
0.0002 100000.       parameters for C3AH6 nucleation
0.2 2500.            parameters for FH3 nucleation
20                   frequency in cycles to check pore space percolation
500                  frequency in cycles to check total solids percolation
11.5                 induction time in hours
22.0                 initial temperature in degrees Celsius
36.5                 activation energy in kJ/mole
0.00134              conversion factor to go from cycles to time
0.72                 mass fraction of aggregates in mixture proportions
0                    flag indicating hydration is under isothermal conditions
```