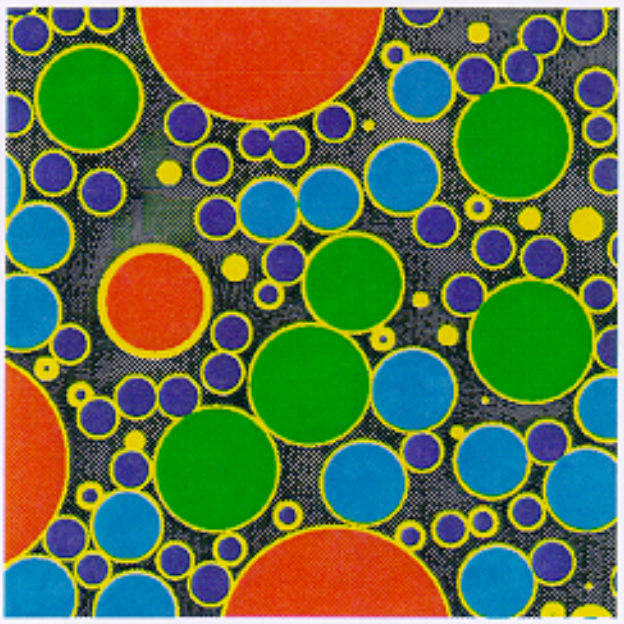
NISTIR 6265

A Hard Core/Soft Shell Microstructural Model for Studying Percolation and Transport in Three-Dimensional Composite Media

Dale P. Bentz
Edward J. Garboczi
Kenneth A. Snyder



Building and Fire Research Laboratory
Gaithersburg, Maryland 20899

NIST

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

NISTIR 6265

A Hard Core/Soft Shell Microstructural Model for Studying Percolation and Transport in Three-Dimensional Composite Media

Dale P. Bentz
Edward J. Garboczi
Kenneth A. Snyder

January 1999
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
Gary R. Bachula, *Under Secretary for Technology*
National Institute of Standards and Technology
Raymond G. Kammer, *Director*

BLANK PAGE

ABSTRACT

This user's manual provides documentation and computer program listings for the three-dimensional hard core/soft shell microstructure model developed at the National Institute of Standards and Technology. The model has been developed to represent the three phase microstructure of a concrete composite consisting of a bulk matrix (cement paste), hard core particles located at random in the bulk (aggregates), and a concentric soft shell surrounding each hard core particle (interfacial transition zone paste). However, this microstructural model is generic in nature and has also been applied to modelling air voids in concrete, macro-defect-free cements, and silica sol gels. Any system consisting of partially overlapping or totally non-overlapping spherical particles can be modelled using the developed computer programs, perhaps with some minor modifications. In addition to generating a representative 3-D microstructure, the programs can also be used to assess the percolation characteristics of the microstructure and estimate a diffusivity (or electrical or thermal conductivity) for the composite media. 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, computer modelling, concrete, diffusivity, hard core soft shell, microstructure, percolation, simulation.

BLANK PAGE

Contents

List of Figures

1 Introduction

A package of computer programs for simulating the microstructure of a three-dimensional cubic volume of concrete has been developed. A three-dimensional representation of microstructure is necessary for the computation of percolation and physical properties for comparison to experiment. Concrete nominally consists of air voids and aggregates (rocks and sand) dispersed in cement paste (cement + water + admixtures). However, its microstructure is complicated by the presence of an interfacial transition zone (ITZ) present at each aggregate (air void)-cement paste interface. The ITZ cement paste generally has a different microstructure and different material properties than the bulk paste [1, 2, 3]. Fortunately, the literature contains a composite material model, the so-called hard core/soft shell (HCSS) model [4, 5], which is a direct analogy to the concrete system. This model consists of hard core particles located in a homogeneous matrix, each particle being surrounded by a concentric soft shell (equivalent to the ITZ regions). The hard core particles can not overlap one another, but the soft shells are allowed to overlap each other and the hard core particles.

At NIST, we have developed a set of computer programs to simulate the microstructure of a 3-D cubic volume of hard core/soft shell particles randomly placed in a bulk matrix. While developed specifically to model the microstructure of concrete [6, 7, 8, 9], these programs have also found ready application to macro-defect-free cements [10, 11], analysis of air void distributions in concrete [12], and nanometer-scale modelling of the calcium silicate hydrate gel present in cement paste [13]. These programs will be described in detail in section 2 of this user's manual and example results will be provided in section 3. Potential program modifications are presented in section 4, and actual code listings and example datafiles are provided in the appendices.

The computer codes are available via anonymous ftp as part of the Computer Integrated Knowledge System for High-Performance Concrete [14] being developed as part of the Partnership for High Performance Concrete Technology program [15] in the Building and Fire Research Laboratory at the NIST. They may be accessed in the `/ftp/pub/HCSSMODEL` 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 and pdf versions of this manual are also available in a subdirectory, `/ftp/pub/HCSSMODEL/manual`.

2 Model Description

The basic characteristics of the HCSS model are outlined in Figure 1. The 3-D microstructure consists of a set of randomly located (centroid=(x_i, y_i, z_i)) hard core spherical particles (radius r_i following a user-selected particle size distribution), each surrounded by a concentric "soft" shell of thickness d_i. For the codes described in this report, d_i is a constant value for all particles in the system (not a function of radius), although it can be easily made to be variable, as will be presented in section 4. The hard core particles are not allowed to overlap during the particle placement process, but the soft shells may overlap one another and partially or totally overlap the hard core of a different particle. The 3-D microstructure is arbitrarily set to be of a size of 100.0 units in each dimension via the parameter **SYSIZE** defined at the beginning of **concrete.c**. The equivalent real system size (such as 10 mm or

100 μm) is defined by the system parameter **REALSIZE**. For a typical concrete or mortar application, the real system size is between 10 mm and 30 mm on a side.

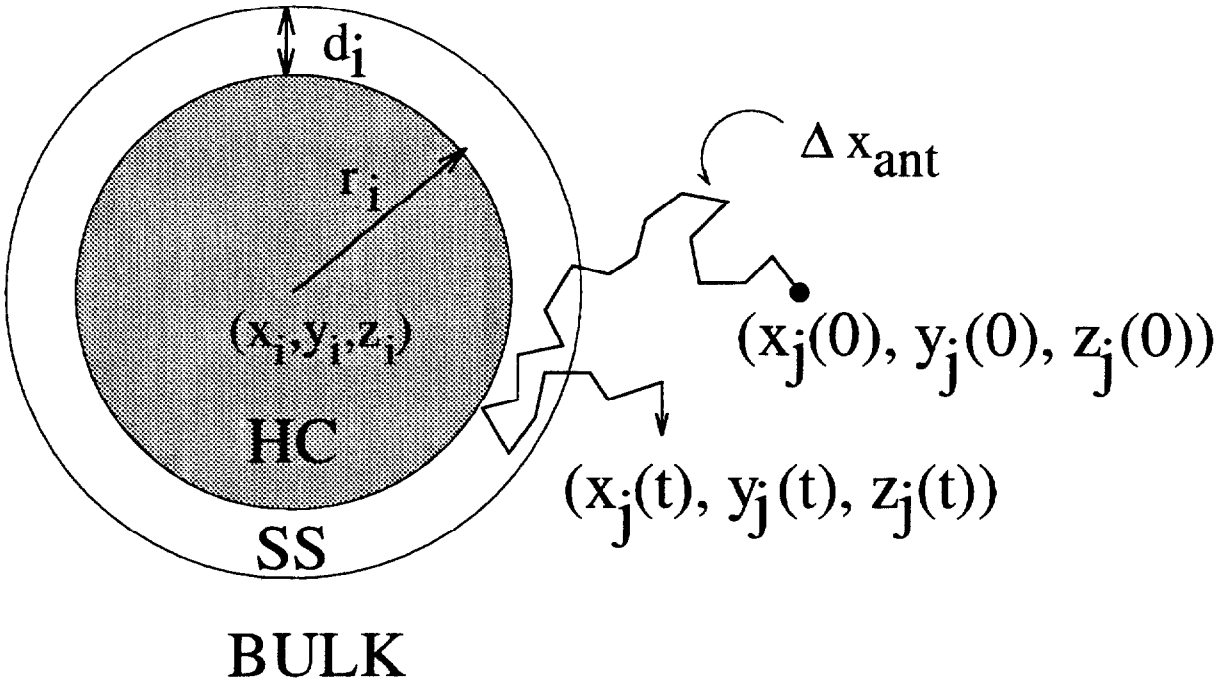


Figure 1: Schematic of a single HCSS particle centered at (x_i, y_i, z_i) and one diffusing ant undergoing a random walk from $(x_j(0), y_j(0), z_j(0))$ to $(x_j(t), y_j(t), z_j(t))$, in steps of size Δx_{ant}.

Because we are also interested in assessing the transport properties of these microstructures, Fig. 1 illustrates the random walk process used to estimate the diffusivity of the composite media. Randomly located "ants" take random steps of size Δx_{ant}, starting from position $(x_j(0), y_j(0), z_j(0))$. Each ant keeps a record of its starting location, its current location, and the time, t_j, elapsed on its personal clock. Special rules are used when the ant's random step will take it across a bulk matrix-soft shell boundary, as will be described later. After a sufficient number of steps (typically hundreds of thousands), the effective diffusivity of the composite media relative to the value of the bulk phase may be estimated by averaging the quantity R_j^2/t_j over all ants and computing:

$$\frac{D_{eff}}{D_{bulk}} = \frac{\langle \frac{\bar{R_j^2}}{t_j} \rangle}{\Delta x_{ant}^2} * (1. - V_{HC}) \tag{1}$$

where t_j is the elapsed time for this ant, V_{HC} is the volume fraction of hard core particles, and R_j for each ant is given by:

$$R_j(t_j) = \sqrt{(x_j(t_j) - x_j(0))^2 + (y_j(t_j) - y_j(0))^2 + (z_j(t_j) - z_j(0))^2} \tag{2}$$

with $(x_j(t_j), y_j(t_j), z_j(t_j))$ being the current location of the ant at time t_j. D_{eff} is the diffusivity of an ion through the composite media and D_{bulk} is its value in the bulk matrix

phase. The outlined techniques can also be applied to electrical conduction and heat transfer problems as well as the diffusion analysis developed here. Because of the mathematical equivalence of Fick's 1st law, Fourier's law, and Ohm's law, we have:

$$\frac{D}{D_{bulk}} = \frac{\sigma}{\sigma_{bulk}} = \frac{k}{k_{bulk}} \tag{3}$$

where σ and k represent electrical and thermal conductivity, respectively. For the codes developed here, we are assuming that the hard core particles are non-conducting/insulative (i.e., D_{HC}=σ_{HC}= k_{HC}=0), such that the "ants" are prohibited from entering the interior of the hard core particles as illustrated in Fig. 1. However, this condition can be easily relaxed.

2.1 Data structures

For computational efficiency, the 3-D computational volume is subdivided into a cubic array of bins (e.g., 30x30x30). The number of bins in each direction is specified by the global parameter **NBIN** in **concrete.c**. The use of bins greatly speeds up execution of the program, as we can limit particle overlap checks to a subset of the overall system. With this in mind, our data structures have been established to provide convenient answers to the following two questions: 1) In which bins is particle i? and 2) Which particles are in bin j? Using linked list data structures in the C programming language, we have created a bin list for each particle, and for each bin, a corresponding particle list, as illustrated in Fig. 2. Our basic data structure for a particle (list element) in a bin then consists of the following:

particle ID

radius

(x, y, z) of centroid

pointer to next particle.

In addition, for each particle, we maintain a bin list and a burnt flag indicating if the particle is part of a percolated pathway through the microstructure. The exact definitions of each of these structures can be found in the computer code provided in Appendix A.

2.2 Particle placement

The first step in the computer program is to place the hard core particles at random locations in the 3-D cubic volume. The particles are placed from largest to smallest and are not allowed to overlap. The placement is ordered from largest to smallest to assure that the larger particles can be placed; if the placement order were totally at random, there often would not be any remaining spaces for the placement of a larger particle after many of the smaller particles had been (randomly) placed. While this placement is different from placing the particles randomly according to equilibrium statistics [16], the resultant soft shell volumes are in excellent agreement with the predictions based on equilibrium statistics [17]. During this placement procedure, the data structures for the particle and bin lists are established. Periodic boundaries are maintained during the placement process, such that if a portion of a particle extends across one or more faces of the 3-D volume, it is completed on the opposing sides. For the x and y directions, no extra particles are created to handle the periodic boundary conditions, with the appropriate opposing face bins being added to

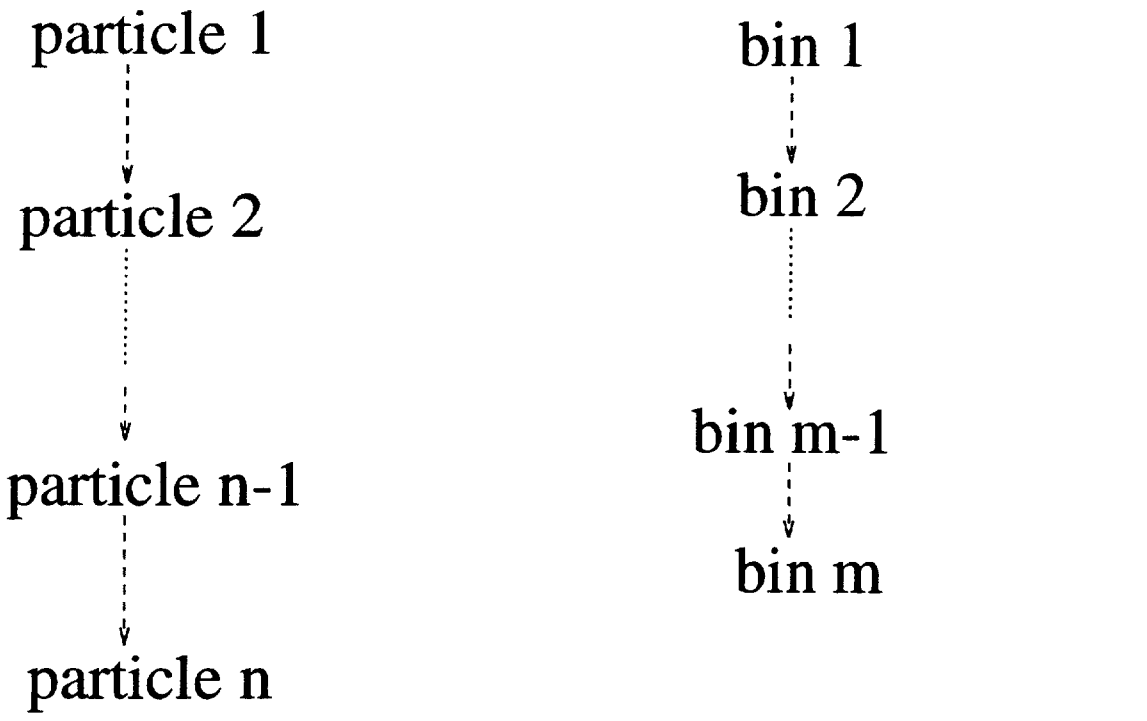


Figure 2: Bin particle list and particle bin list data structures.

the particle's bin list, etc. However, for the z direction, specialized placement conditions are employed, such that a new "virtual" particle is created in addition to the primary particle. This is necessary due to the subsequent assessment of percolation in the z direction, so that a burnt particle at the top surface, which also overlaps the bottom surface, does not constitute a percolated pathway through the system, via only the periodic boundary conditions.

The general algorithm for particle placement is as follows:

```
for each particle to be generated

   set radius (largest to smallest)

   while particle not yet placed

      generate random x,y,z for centroid (each from 0.0 to SYSIZE)

      determine range of bins intersected by the new particle

      for each bin containing particle (and no overlap yet found)

         for each particle already in this bin (and no overlap yet found)
```

```
            check for overlap based on separation distance of
            centroids and the two particle radii

          next particle

        next bin

        if no overlap is found, place particle
        including specialized placement conditions if particle
        overlaps z=0 or z=SYSIZE plane

          determine all bins intersected by particle (including its shell)

          update the particle bin list and bin particle list data structures

    next particle
```

The user has three choices for the radii of the hard core particles. They may be chosen from three distributions: 1) monosize, 2) a Weibull distribution, or 3) a sieve analysis classification. For the Weibull particle size distribution, uniform random radii between 0.0 and 1.0 are first generated, sorted from largest to smallest, and then transformed to Weibull variables as outlined by Law and Kelton [18], using the user-supplied distribution parameters. For the sieve analysis, a supplementary datafile (**sieve.dat**, example provided in Appendix B) is consulted to determine the number of different sieve classes, and the minimum and maximum diameter and **number fraction** for each sieve class. The particles are distributed uniformly by volume within each sieve class. The program **randnum.c** provided in Appendix A generates the actual particle radii for the latter two choices, although the transformation to Weibull variables is executed in a routine provided in the main **concrete.c** program. If the user wanted to incorporate another particle size distribution (lognormal, etc.), they would simply need to modify this Weibull distribution generating routine (**radgen**) in the main program and change the input statement to reflect the new parameters required for their desired distribution.

2.3 Percolation assessment

Our goal in percolation assessment is to determine if a connected pathway exists across the microstructure (from $z = 0.0$ to $z = 100.0$) staying entirely within the soft shells. If such a path exists, we mark all particles that are a part of it so that we may later determine the volume fractions of hard cores and soft shells which are "percolated." Our algorithm is a continuum variation of the 'burning" algorithm conventionally used for assessing the percolation properties of digital images [19]. The general algorithm for performing this percolation assessment, implemented in **burnt.c** of Appendix A is as follows:

```
initialize counters for percolation to zero

for each bin on the top surface (zbin=0)

   for each particle in bin
```

```
      determine if particle overlaps the z=0 plane (including its shell)
      determine if the particle is not yet marked as burnt
      if not yet burnt, set old particle burnt list to this particle
      set particle's burnt flag to burnt (2)
      add particle to total burnt list
      while there are particles in old burnt list
         for each particle in old burnt list
            determine all bins intersected by this particle
            for each bin this particle is part of
               for each other particle in this bin
                  if overlap of soft shells and new particle is not yet
                  marked as burnt
                     add particle to new burnt list
                     add particle to total burnt list
                     set its burnt flag to 1
               next other particle in bin
            next bin contacted by the primary particle
         next primary particle in old burnt list
         copy new burnt list to old burnt list
      end of a percolated structure (new burnt list is empty)
      check if z=SYSIZE plane is intersected by a particle
      in the total burnt list and update counters (percolation)
   next particle in this top level bin
next bin on the top surface
output results
```

2.4 Volume fraction via systematic point sampling

Once the particles have been placed and percolation assessed, the next step is to determine the volume fractions of the various phases and their connected (percolated) volume fractions as performed in the program **volume.c** provided in Appendix A. Of course, for the hard cores, the volume fraction is simply the sum of the volume of each particle, divided by the system volume. However, because of overlaps, the volume of the soft shells is much more difficult to determine analytically, although analytical equations developed by Lu and Torquato [16] have been shown to be very accurate [12]. For our purposes, to determine the overall and percolated volume fractions of the hard core and soft shell phases, systematic point sampling is used. The cubic volume is sampled at a fairly high resolution (100^3 = 1,000,000 points being typical), and the phase present at each point is determined. This collection of points is then averaged to determine the volume fractions. For random point sampling, based on formulas provided by Cochran [20], this level of random point sampling would allow estimation of phase volume fractions to within 2%, even for phase volumes on the order of 0.01. For randomly located objects, systematic point sampling is generally two to four times more efficient than random point sampling [21]. Indeed, for hard core phases present at a volume fraction on the order of 0.005, the point sampling estimated value has been found to still be within 2% of the analytical value. During point sampling, each point is classified as one of the following: hard core in a particle which is part of a percolated pathway, hard core in a non-percolated particle, soft shell in a percolated pathway, soft shell in a pathway accessible from the top of the microstructure but not percolated, soft shell at a location inaccessible from the top of the microstructure, or bulk matrix (outside of all hard cores and shells).

If the user wanted to have a digitized version of the microstructure available for subsequent computations using finite element or finite difference codes [22], this routine could be easily modified to output a value for each of the "pixels" sampled. The nesting of the loops for the bins and sampling points has been set up to facilitate the output of the microstructure, if desired. Thus, the point sampling loops for the z and y directions are placed immediately after their corresponding bin loops. In this way, an output list of the pixels will already be in the appropriate order without any further modification. In other words, in a straight output list of all of the sampled "pixels", the x index will vary first, followed by the y index, and finally the z index. Thus, any program for further processing of the output microstructure only need read in the microstructure using a simple 3-level loop structure (e.g., for z=0 to 100, for y=0 to 100, for x=0 to 100).

The general point sampling algorithm is as follows:

```
initialize counters for phases to zero

for each bin in the z direction (zbin)

for each point in the z direction in zbin (Z)

   for each bin in the y direction (ybin)

   for each point in the y direction in ybin (Y)

      for each bin in the x direction (xbin)
```

```
        for each point in the x direction in xbin (X)

          for each particle in bin (xbin,ybin,zbin) and point (X,Y,Z)
          not yet in hard core

             if (X,Y,Z) is inside particle, set hard core flag

             if (X,Y,Z) is inside particle soft shell, set soft shell flag

          next particle in bin

          update counters based on hard core, soft shell, and particle
          burnt flags

        next X

        next xbin

    next Y

    next ybin

next Z

next zbin

output results
```

2.5 Diffusivity estimation via myopic random walkers

As mentioned in the previous section, the 3-D microstructure could be easily digitized and finite difference or finite element techniques applied to determine transport and mechanical properties. The basic problem is one of resolution. Because the soft shell thickness is typically one order of magnitude, or more, smaller than the particle radii (e.g., 25 μm vs. 1 mm for a sand particle in concrete), to adequately consider both of these phases and, more importantly, their connectivity in a digitized microstructure requires an extremely large system size. For this reason, continuum random walker techniques to estimate transport properties have been developed.

The techniques are described in detail elsewhere [7, 8, 23]. Special rules are used when the myopic random walkers (ants) attempt to cross from the bulk matrix to the soft shell, based on the ratio of the soft shell diffusivity to that of the bulk matrix, a parameter provided as input by the user. In addition, the time a given unit step takes in a phase is inversely proportional to the relative diffusivity of the phase. For example, a step in the more conductive soft shell requires less time than one of the equivalent distance in the bulk matrix. For this reason, each ant carries its own personal clock throughout the simulation. The user supplies the step size to be used, typically set at a value of $\frac{1}{3}$ of the soft shell thickness. If the step size is too large, the ants will step from the bulk matrix and contact

the hard cores (reflecting back to their former position) without appropriately sampling the soft shell volume. In essence, in this case, our continuum simulations will suffer from the same resolution dilemna as that described above for the digitized systems. The ratio of the step size to the system size in the continuum microstructure plays a similar role as the ratio of the pixel length to the system size in a digitized microstructure.

The user also inputs the number of requested total steps, n_{steps}, for each ant and the program output results after $\frac{n_{steps}}{4}$, $\frac{n_{steps}}{2}$, and n_{steps} steps. The output file consists of a list (one entry per ant labelled j) of the current distance squared, R_j^2, the ant has travelled and the current elapsed time, t_j, on its clock. This file can then be analyzed, using a spreadsheet for example, to divide the R_j^2 values by their corresponding t_j values, to determine the average value of R_j^2/t_j, and equation 1 employed to estimate the value of $D_{\text{eff}}/D_{\text{bulk}}$.

The general algorithm for executing the myopic random walks is as follows:

```
place ants in 3-D volume avoiding hard cores and initialize clocks

for each requested step

   for each placed ant

      calculate new random position point on a sphere of radius= step size
      centered at current ant position

      determine bin of new position point

      for each particle in this bin and point not yet in hard core

         check if point is in hard core or soft shell

      next particle

      if not in hard core and move is probable

         update position in both periodic (0 to SYSIZE) and
         absolute coordinates

         update phase ID based on new position of ant (bulk or soft shell)

      update clock

   next ant

if (1/4), (1/2) or all steps taken, output results to file

next step
```

2.6 Input datafile description

Sample input datafiles are provided in Appendix B. The input parameters necessary for a single execution of the program are as follows:

random number seed- negative integer

number of particles to place

particle type choice (0=monosize, 1=Weibull, 2=sieve analysis)

if particle type=0, the radius of the monosize particles in **system units**

if particle type=1, the parameters (α, β, γ) of the Weibull distribution in **system units**

the thickness of the soft shells in **system units**

the resolution (sample points per bin dimension) for the volume sampling

the ratio of the diffusivity of the soft shells to the bulk matrix

the step size for the random walkers in **system units**

the number of steps for each random walker to take.

These input values are echoed to the standard output file (which the user may redirect to the file of their choice) for a permanent record of the input stream for each execution of the program. A typical compile/run sequence on a UNIX workstation might be:

```
cc concrete.c -o concrete -lm -O2
concrete <concrun.dat >concrun.out
```

where **concrun.dat** contains the input data and **concrun.out** will hold the results generated by the computer program.

If the user selects particle type=2, an auxiliary datafile is consulted to provide the detailed information for the sieve analysis for the aggregates. An example sieve datafile (**sieve.dat**) is also provided in Appendix B. It simply contains the number of separate sieves, followed by a listing for each sieve range (going from largest to smallest in diameter). Each row contains the minimum and maximum diameters (in **real physical units** such as mm or cm) and the **number** fraction of particles for that sieve class. If the particle diameters are expressed in mm, the user must be sure to express the parameter **REALSIZE** in mm as well.

3 Example Results

The computer code described herein has been used extensively at NIST to simulate a variety of cement-based systems [6-13]. A brief survey of various results generated using the computer codes will be presented here.

The most basic application of the code would be to a system of monosize spheres, where the ratio of the soft shell thickness to the hard core radius is systematically varied. Figure 3 shows the results obtained for the volume fractions of hard cores and soft shells needed to just form a percolated pathway across the microstructure. For the case of totally overlapping particles (i.e., hard core radius=0), the classical result of 0.29 volume fraction is obtained

[7]. As the ratio of hard core to total radius increases, less soft shell volume is needed to form a percolated pathway. For comparison, for a mortar (sand and cement paste), typically a volume fraction of soft shells on the order of 0.10 is needed to achieve percolation [24].

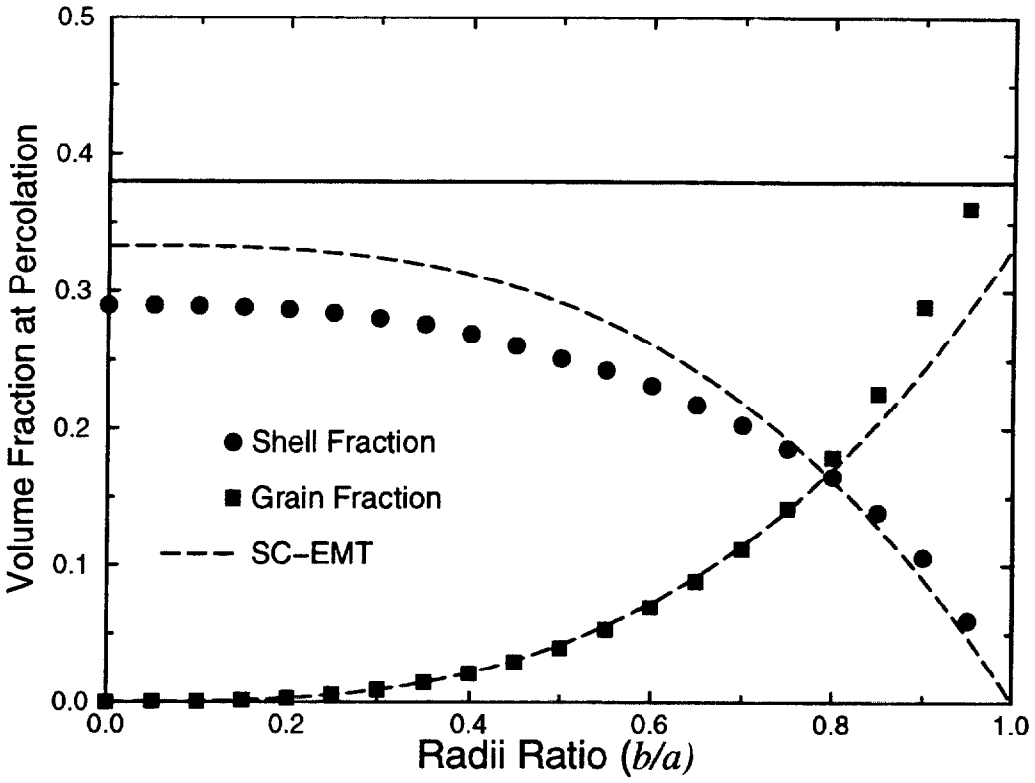


Figure 3: Computer model hard core soft shell percolation results for monosize spheres with variable ratios of hard core radius b to total (hard core + soft shell) radius a. The data points indicate the volume fractions of hard core grains and soft shells needed for percolation and the dashed lines indicate the predicted results of a self-consistent effective medium theory (SC-EMT) [7]. The solid horizontal line indicates the random parking limit for monosize spheres (about 0.38).

In another study [6], the HCSS model was applied to determine the best value for the thickness of the ITZ regions to match mercury intrusion porosimetry (MIP) experimental results. In this study, a series of mortars of variable hard core volume fraction were prepared and examined using the MIP technique. As the hard core volume fraction increased, a point was reached where the soft shells (which contain more and larger pores) achieved percolation. In MIP, this percolation was indicated by the appearance of a new class of pores of larger size than those determined for the bulk paste. Since the size distribution of the sand (hard cores) was measured experimentally, the simulation was conducted matching the sieve classification and varying the thickness of the soft shells. The results presented in Fig. 4 show the connected fraction of soft shells plotted vs. the volume fraction of hard core particles. Experimentally, the percolation was observed to occur between 45 % and 49 % sand volume fraction. From Fig. 4, this would best correspond to an ITZ (soft shell) thickness of 15 μm to 20 μm, in general agreement with experimental microscopy observations [2, 3].

A further issue which can be addressed by varying the thickness of the soft shells is the determination of the volume fraction of matrix within a given distance of the hard core

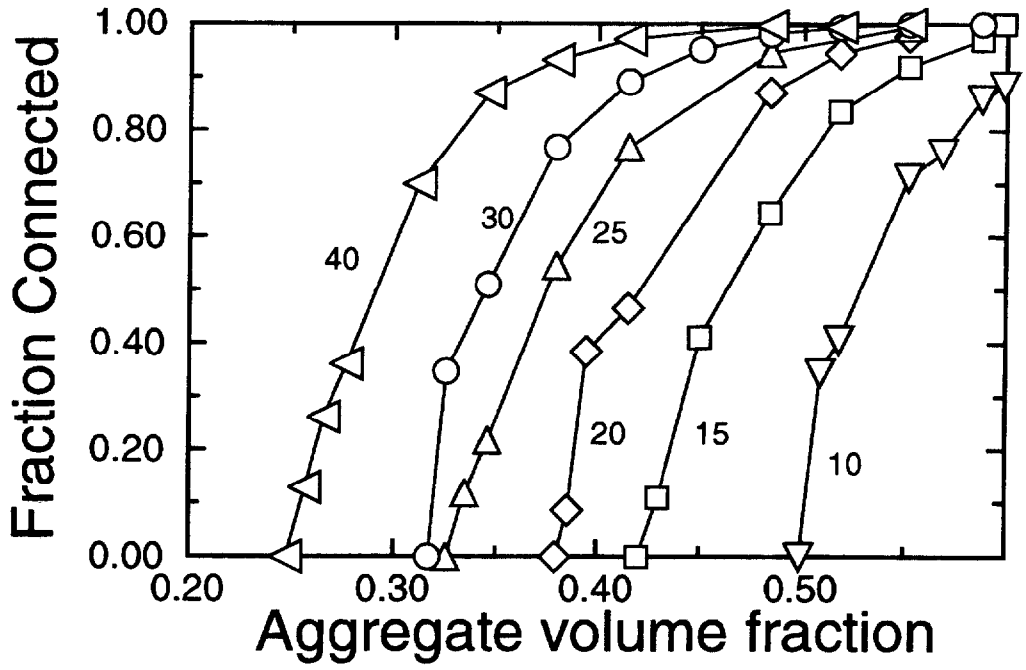


Figure 4: Computer model interfacial zone percolation results for varying sand volume fractions and interfacial zone thicknesses. Labels provide the interfacial transition zone thicknesses in μm.

particles. Using realistic size distributions for the aggregate particles, Fig. 5 provides a plot of the percentage of the cement paste within a given distance of an aggregate for both a typical mortar and a typical concrete. Because of the higher surface area of the aggregate particles in a mortar relative to those in a concrete, a larger volume fraction of paste is found within a fixed distance of the aggregate for the mortar. Given an ITZ thickness on the order of 20 μm, one finds that an appreciable volume fraction of the cement paste (on the order of 20 %) is contained in the ITZ regions, which could have a significant influence on transport and mechanical properties. The equations of Lu and Torquato [16] can also be used to very accurately predict these volume fractions of paste within a given distance of an aggregate [17].

This concept can be extended in the case of air voids in concrete to consider the distribution of void to void distances [12]. Figure 6 provides a plot of the mean void-void spacing as a function of the void radius (for voids of radius r, what is the average surface to surface distance to the nearest neighbor void). It can be seen that the larger the void radius, the more likely that there exists another void within a short distance of its surface. The predictions of Lu and Torquato [16] are seen to be in excellent agreement with the simulation results.

The random walker diffusivity code has been used to predict the diffusivity of concrete as a function of mixture proportions [9] and also to evaluate the diffusivity of a series of mortars of variable sand contents [26]. Figure 7 shows a comparison of experimental and model results for a series of mortars prepared using a water-to-cement (w/c) ratio of 0.5. In general, the agreement between experimental and model values is quite reasonable, as the

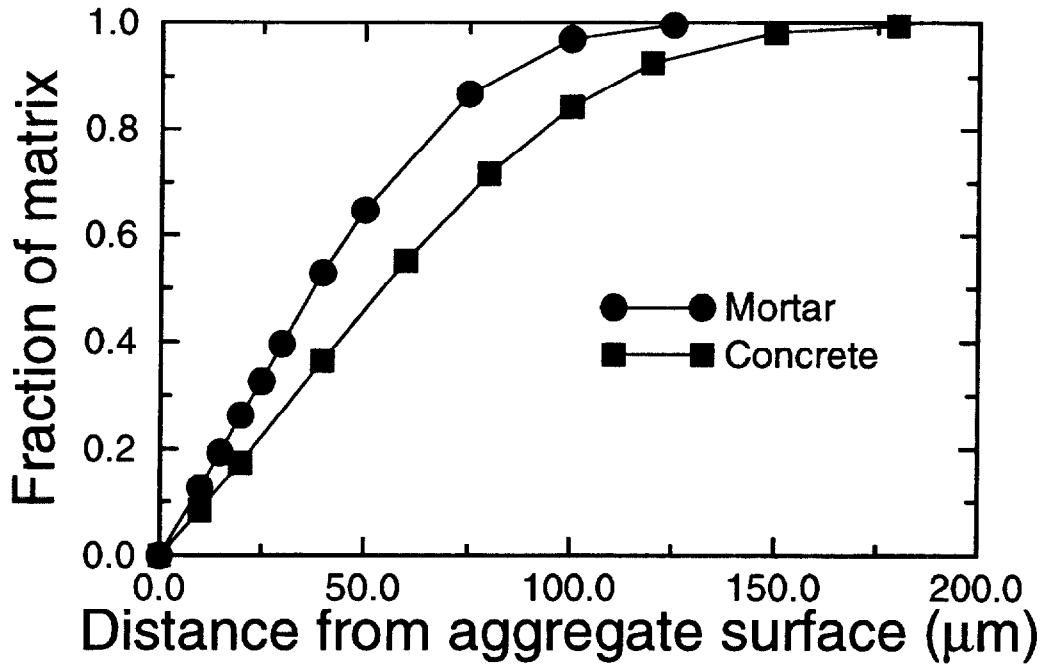


Figure 5: Fraction of total cement paste within a given distance of aggregate surface, for a mortar (V_{HC}=0.552) and for a concrete (V_{HC}=0.646).

difficulties in making accurate experimental measurements usually result in a coefficient of variation on the order of 20 %.

4 Potential Modifications to Code

The code has been developed in a modular fashion so that the user can quickly and efficiently make modifications. As will be illustrated in the following examples, typically, the user may only need to change one of the major program routines (placement, percolation, volume, or diffusion) to implement a modification. The examples provided below are meant to give a brief idea of some potential modifications, and are by no means exhaustive. The code has been constructed in a flexible framework, and its adaptation and extension are hopefully only limited by the creativity and imagination of the user.

4.1 No shell or variable shell thickness

For applications where a two phase (hard core and bulk) system is sufficient, the thickness of the soft shells can be easily set to zero. However, if desired, further improvements in computational efficiency could be obtained by carefully eliminating all references to the soft shells from the computer code. For a variable shell thickness, the shell thickness can be incorporated into the particle data structure, so that the radius and shell thickness of each particle are known. This has been implemented previously for the simple case of a mixture of particles, some with a fixed thickness shell and others with no shell (inert), to simulate

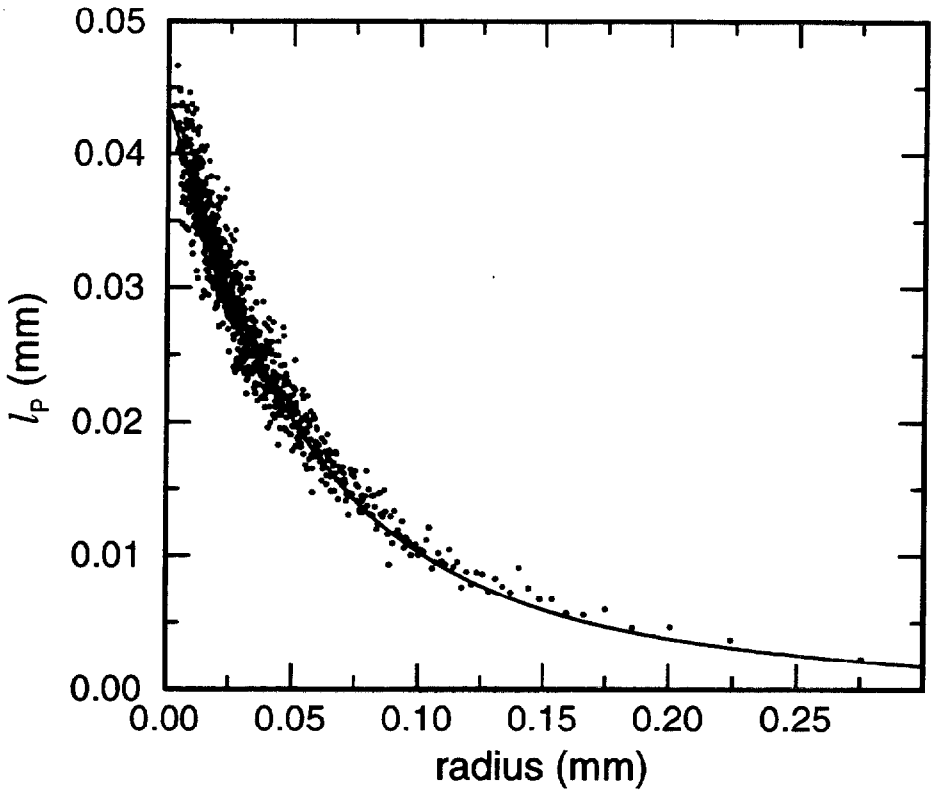


Figure 6: Mean void-void spacing (l_p) for lognormally distributed sphere radii with a density of 240 mm^{-3} [12]. Measured values are shown as solid circles; the solid line is the estimate of Lu and Torquato [16].

macro-defect-free cements containing a mixture of reactive cement and inert Al_2O_3 particles [11]. Conversely, freely overlapping monosize particles can be easily generated with the current code by simply setting the radius of monosize spheres to zero.

4.2 Clustering of particles

For modelling silica and calcium silicate hydrate gels [13], some clustering of the particles may be needed. One way to cluster the particles could be to modify the particle placement code in **concrete.c** to place n_{seed} particles totally at random, and then assuring that all subsequent particles have a soft shell that overlaps at least one other existing soft shell. The n_{seed} particles thus serve as "seeds" around which the subsequently placed particles will cluster.

4.3 Ellipsoidal particles

Assessing the overlap characteristics of two general ellipsoidal particles is more complex and requires much more computer time than determining the overlap of two simple spheres. Nevertheless, the general algorithms to be used are the same as those outlined in section 2. The authors have developed programs to model microstructures based on general three-dimensional ellipsoidal particles as opposed to spheres [24]. In this case, the data structure

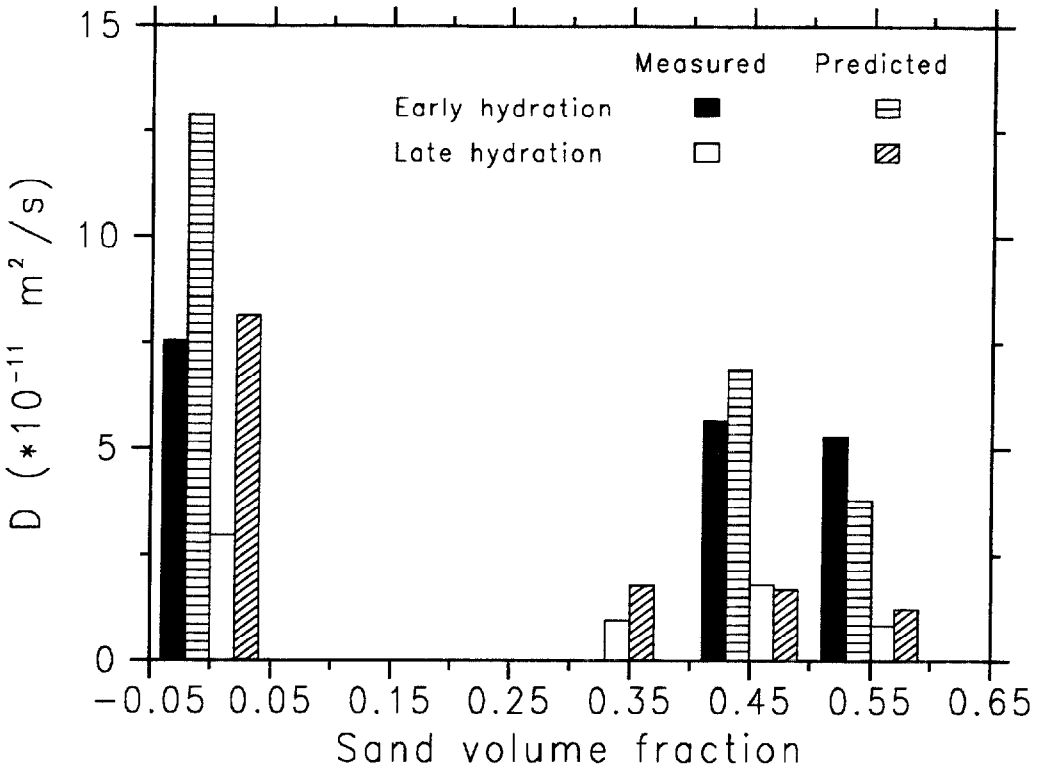


Figure 7: Comparison of experimental and model diffusion coefficients for w/c=0.50 [26].

for each particle in a bin includes three principal radii and three angles of orientation, as well as the centroid location. The authors may be contacted for further details or if someone wishes to obtain a copy of the ellipsoid codes. Basically, for the concrete system, ellipsoidal particles result in a lower hard core particle volume being necessary to achieve percolation of the soft shells, due to the increased surface area of an ellipsoid relative to a sphere of equivalent volume [24].

4.4 First passage times in random walks

For the random walkers, a second problem of interest is that of the first passage time distribution, or the time needed for randomly located walkers to first contact a hard core surface. This has implications in NMR relaxation data obtained on real microstructures [25]. To facilitate this simulation, an active flag has been included as part of the data structure for each random walker. To implement a first passage time problem, the user simply needs to deactivate each walker the first time their random step takes them into a hard core particle and to output the corresponding time on the walker's clock to a file. In this case, instead of using the number of steps input by the user, the code could be modified to continue the random walks until all (or some fixed percentage) of the ants are deactivated, by adding a flag variable that indicates whether one or more ants (or how many ants) are still active.

5 Summary

A package of programs for simulating a hard core/soft shell microstructure in three dimensions has been presented. The programs allow for random particle placement, percolation assessment, volume fraction point sampling, and diffusivity estimation. The code has been developed in the C programming language, using linked list data structures and a modular program structure, to facilitate future modifications by the research community. The applications of the program to modelling the microstructure of cement-based materials are numerous and the programs should ultimately find applications to a diverse population of material systems.

6 Acknowledgements

The authors would like to acknowledge useful discussions with Dr. Larry Schwartz of Schlumberger and Prof. Sal Torquato of Princeton University. The authors would also like to acknowledge the NSF Center for Advanced Cement-Based Materials (ACBM) and the NIST Partnership for High Performance Concrete Technology program for partial funding of this software development and the preparation of this documentation.

7 References

[1] 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, Ed. J.P. Skalny and S. Mindess (American Ceramic Society, Westerville, OH, 1995), pp. 155-200.

[2] Interfaces in Cementitious Composites, Ed. J.C. Maso, Vol. **18**, E & FN Spon, London, 1992.

[3] Interfacial Transition Zone in Concrete, Ed. J.C. Maso, E & FN Spon, London, 1996.

[4] Rikvold, P.A., and Stell, G., *Journal of Colloid and Interface Science*, **108** (1), 158-173, 1985.

[5] Lee, S.B., and Torquato, S., *Journal of Chemical Physics*, **89** (5), 3258-3263, 1988.

[6] Winslow, D.N., Cohen, M.D., Bentz, D.P., Snyder, K.A., and Garboczi, E.J., *Cement and Concrete Research*, **24**, 25-37, 1994.

[7] Schwartz, L.M., Garboczi, E.J., and Bentz, D.P., *Journal of Applied Physics*, **78** (10), 5898-5908, 1995.

[8] Garboczi, E.J., Schwartz, L.M., and Bentz, D.P., *Advanced Cement-Based Materials*, **2**, 169-181, 1995.

[9] Bentz, D.P., Garboczi, E.J., and Lagergren, E.S., *Cement, Concrete, and Aggregates*, **20** (1), 129-139, 1998.

[10] Lewis, J.A., Boyer, M., and Bentz, D.P., *Journal of the American Ceramic Society*, **77** (3), 711-716, 1994.

[11] Desai, P., Lewis, J.A., and Bentz, D.P., *Journal of Materials Science*, **29** (4), 6445-6452, 1994.

[12] Snyder, K.A., *Advanced Cement-Based Materials*, **8**, 28-44, 1998.

[13] Bentz, D.P., Quenard, D.A., Baroghel-Bouny, V., Garboczi, E.J., and Jennings, H.M., *Materials and Structures*, **28**, 450-458, 1995.

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

[15] Frohnsdorff, G., "Partnership for High-Performance Concrete," in Proceedings of the International Symposium on High-Performance and Reactive Powder Concretes, Ed. P.C. Aitcin, 51-73, 1998.

[16] Lu, B., and Torquato, S., *Physical Review A*, **45**, 5530-5544, 1992.

[17] Garboczi, E.J., and Bentz, D.P., *Advanced Cement-Based Materials*, **6**, 99-108, 1997.

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

[19] Stauffer, D., and Aharony, A., Introduction to Percolation Theory, 2nd Ed. (Taylor and Francis, London, 1992).

[20] Cochran, W.G., Sampling Techniques (Wiley, New York, 1981).

[21] Bentz, D.P., and Martin, J.W., *Journal of Coatings Technology*, **57** (726) 43-49, 1985.

[22] Garboczi, E.J., "Finite Element and Finite Difference Programs for Computing the Linear Electrical and Elastic Properties of Digital Images of Random Materials," NISTIR **6269**, U.S. Department of Commerce, December 1998. Also available at http://ciks.cbt.nist.gov/garboczi/ in Chapter 2.

[23] Hong, D.C., Stanley, H.E., Coniglio, A., and Bunde, A., *Physical Review B*, **33** (7), 4564-4573, 1986.

[24] Bentz, D.P., Hwang, J.T.G., Hagwood, C., Garboczi, E.J., Snyder, K.A., Buenfeld, N., and Scrivener, K.L, *MRS Symposium Proceedings*, **370**, 437-442, 1995.

[25] Straley, C., and Schwartz, L.M., *Magnetic Resonance Imaging*, **14** (7-8), 999-1002, 1996.

[26] Bentz, D.P., Detwiler, R.J., Garboczi, E.J., Halamickova, P., and Schwartz, L.M., "Multi-Scale Modelling of the Diffusivity of Mortar and Concrete," in Chloride Penetration Into Concrete, Ed. L.O. Nilsson, 85-94, 1997.

Appendices

A Computer programs for hard core/soft shell model

A.1 Listing for randnum.c

```
/*************************************************************************/
/*                                                                       */
/*   Program: randnum.c                                                  */
/*   Purpose: To generate random radii for particles sorted from         */
/*            largest to smallest                                        */
/*   Contains: genrand (random radii from 0.0 to 1.0)                    */
/*             gensieve (radii based on sieve size classes)              */
/*   Programmer: Dale P. Bentz                                           */
/*               NIST                                                    */
/*               Building 226 Room B-350                                 */
/*               Gaithersburg, MD  20899-8621                            */
/*               Phone: (301) 975-5865  Fax: (301) 990-6891              */
/*               E-mail: dale.bentz@nist.gov                             */
/*               WWW page: http://titan.cbt.nist.gov/bentz               */
/*                                                                       */
/*************************************************************************/
/* Uniform random deviates can be transformed to other */
/* distributions (Weibull, etc.) as needed */
/* nsort is number of particle radii to be generated */
void genrand(nsort)
     int nsort;
{
     int ic,jc;       /* loop index variables */
     float temp;      /* interchange variable */

     /* Generate the random deviates */
     for(ic=0;ic<nsort;ic++){
          randrad[ic]=ran1(seed);
     }

     /* Sort the random deviates from largest to smallest */
     /* for use in particle placement */
     /* Simple bubble sort */
     for(ic=0;ic<(nsort-1);ic++){
     for(jc=(ic+1);jc<nsort;jc++){
          if(randrad[ic]<randrad[jc]){
               temp=randrad[ic];
               randrad[ic]=randrad[jc];
               randrad[jc]=temp;
```

```
        }
    }
    }
}

/* Routine to generate particle radii based on a sieve analysis */
/* nsort is the number of particle radii to generate */
void gensieve(nsort)
    long int nsort;
{
    FILE *fsieve;      /* FILE identifier for sieve file */
    float dlow,dhigh;  /* diameters for an individual sieve class */
    float vlow,vhigh;  /* volumes for an individual sieve class */
    /* number fraction of particles for an individual sieve class */
    float npart,
          step,        /* step size in volume space for a sieve */
          vone,        /* volume of an individual particle */
          rcubed;      /* radius cubed for an individual particle */
    double vtot,stot;  /* total volume and surface areas for system */
    int nsieve,isv;    /* number of sieve classes and loop index */
    long int ngen,     /* counter for number of particles generated */
             itodo,    /* number of particles needed */
             ido;      /* loop index for number of particles needed */

    ngen=0;     /* Counter for number of particles generated */
    vtot=stot=0.0;   /* Sums for volume and surface area counts */
    fsieve=fopen("sieve.dat","r");
    /* Read in the number of sieves being considered */
    fscanf(fsieve,"%d\n",&nsieve);

    /* loop for each sieve */
    for(isv=1;isv<=nsieve;isv++){
        /* Read in bounding diameters and number fraction */
        /* of particles for this sieve */
        fscanf(fsieve,"%f %f %f\n",&dlow,&dhigh,&npart);

        /* Compute particle volumes and distribute requested number */
        /* of particles uniformly in volume space */
        vlow=PIVAL*CUB(dlow)/6.;
        vhigh=PIVAL*CUB(dhigh)/6.;
      printf("Seive %d - Vlow= %f Vhigh= %f frac= %f \n",isv,vlow,vhigh,npart);
        itodo=(int)((float)nsort*npart+0.5);   /* No. of particles needed */
        /* For last sieve, generate all remaining needed particles */
        /* to handle roundoff error, etc. */
        if(isv==nsieve){itodo=nsort-ngen;}
        step=0.0;
```

```
        /* Determine volume step for sequential particles */
        if(itodo>1){step=(vhigh-vlow)/(float)(itodo-1);}

        printf("For particle size %f no. of particles is %ld \n",vlow,itodo);
        fflush(stdout);
        /* loop for all needed particles */
        for(ido=1;ido<=itodo;ido++){
             ngen+=1;
             /* Compute volume of this particle */
             vone=(vhigh-step*(float)(ido-1));
             if(itodo==1){vone=((vhigh+vlow)/2.);}

             /* Convert to radius */
             rcubed=3.*vone/(4.*PIVAL);

             /* Convert to system coordinates */
             randrad[ngen]=(SYSIZE/REALSIZE)*pow(rcubed,(1./3.));

             /* Accumulate volume and surface area counts */
             vtot+=4.*PIVAL*CUB(randrad[ngen])/3.;
             stot+=4.*PIVAL*SQR(randrad[ngen]);
             }
    }

    fclose(fsieve);
    /* Output volume and surface area in total */
    printf("Volume and surface area are %lf and %lf \n",vtot,stot);
    fflush(stdout);
}
```

A.2 Listing for burnt.c

```
/*****************************************************************/
/*                                                               */
/*   Program: burnt.c                                            */
/*   Purpose: To burn 3-D continuum structure (HCSS model)       */
/*            from top to bottom                                 */
/*            Burning is periodic from left to right and back to */
/*              front!                                           */
/*   Programmer: Dale P. Bentz                                   */
/*               NIST                                            */
/*               Building 226 Room B-350                         */
/*               Gaithersburg, MD  20899-8621                    */
/*               Phone: (301) 975-5865  Fax: (301) 990-6891      */
/*               E-mail: dale.bentz@nist.gov                     */
/*               WWW page: http://titan.cbt.nist.gov/bentz       */
/*                                                               */
/*****************************************************************/
void burn(ntotal)
     long int ntotal;   /* ntotal is total number of particles in system */
{
     /* data structure to hold all new particles along the burning front */
     struct burnlist {
          int partnum;        /* particle ID */
          struct burnlist *nextburn;  /* pointer to next particle */
     };

     struct burnlist *oldlist,     /* Current burn list to process */
                     *newlist,     /* New list being generated */
                     *addlist;     /* Element to be added to new list */
     struct burnlist *conlist,
                     *connew;      /* Connected lists for a particle */
     struct partlist *bincur,      /* Pointers to all particles in */
                     *bin1;        /* bin being processed */
     struct aggchar *partcur;      /* Pointer to particle being processed */
     struct binlist *newbin;       /* Pointer to bin list for particle */
                                   /* being processed */

     float vfrac,           /* Hard core volume fraction through the sample */
           vtop,            /* Hard core volume frac. accessible from top */
           testv,           /* Distance between particle surface */
    /* and top of box surface */
           x1,y1,z1,        /* Particle 1 centroid */
           x2,y2,z2,        /* Particle 2 centroid */
           r1,r2;           /* Particle radii */
     float dist1,           /* Particle center to center distance */
           dist2;           /* Sum of particle radii + shells */
     /* Average neighbor statistics variables */
```

```
      /* number of connections, distances, and volumes */
      float aveneigh,avetopn,avedist,disttot;
      float disttotc,vtot,vtotc;
      /* Variables for bin identification */
      int xbin,ybin,zbin;
      int conflag;                    /* Flag indicating connectivity */
      long int itotal;                /* Loop index variable */

      int found,                  /* Flag indicating particle found */
          ix,iy,iz,               /* Loop index variables for bins */
          ix1,iy1,iz1,            /* More loop index variables */
          botflag;                /* Flag indicating bottom of system is reached */

      /* counters for number of particles accessible from top */
      /* and number of particles which are part of a path */
      /* through the 3-D system (from top to bottom) */
      long int nburn,ntop,nthru,ntopc,nthruc,nconnect;
      static long int burnid[NPART];    /* ID list of burnt particles */

      /* Initialize variables */
      ntop=0;
      nthru=0;
      nthruc=0;
      ntopc=0;
      disttotc=0.0;
      vtotc=0.0;

   /* reset all burn flags to 0 for next burn call */
      for(itotal=1;itotal<=ntotal;itotal++){
           particle[itotal]->burnt=0;
      }

  /* process all bins at the top of the 3-D system (z=0) */
      iz=0;
      for(ix=0;ix<NBIN;ix++){
      for(iy=0;iy<NBIN;iy++){
           /* pointer to current bin being processed */
           bincur=bin [ix] [iy] [iz];

      /* loop for all particles in this bin */
      while((bincur!=NULL)&&(bincur->partnum!=(-1))){

           /* determine if current particle extends out the top of */
           /* 3-D system (bin) including the soft shell */
           testv=bincur->zloc-(bincur->size)-shellthick;

           /* check if in contact with top and not yet burnt */
```

```
if((testv<=0.0)&&(particle[bincur->partnum]->burnt==0)){

     /* start a burn front from this particle */
     nburn=1;
     burnid[nburn]=bincur->partnum;
     vtot=(4.*PIVAL/3.0)*CUB(bincur->size);
     nconnect=0;
     disttot=0;
     /* mark particle as burnt */
     /* give special value of 2 because of periodic boundaries */
     particle[bincur->partnum]->burnt=2;  /* special */
     oldlist=(struct burnlist *)malloc(sizeof(struct burnlist));
     oldlist->partnum=bincur->partnum;
     oldlist->nextburn=NULL;

     /* while there are particles in the old list */
     while(oldlist->partnum!=(-1)){

          /* initialize the new list for new burn front */
          newlist=(struct burnlist *)malloc(sizeof(struct burnlist));
          newlist->partnum=(-1);
          newlist->nextburn=NULL;
/* burn from all particles in the old burn list */
/* and add newly found particles to the new list */
while((oldlist!=NULL)&&(oldlist->partnum!=(-1))){

    /* initialize connected list for this particle */
    conlist=(struct burnlist *)malloc(sizeof(struct burnlist));
    conlist->partnum=(-1);
    conlist->nextburn=NULL;

          /* pointer to particle being considered */
          partcur=particle[oldlist->partnum];
          /* pointer to bin list for current particle */
          newbin=partcur->binloc;
     /* process all bins in which current particle exists */
     while(newbin!=NULL){

          found=0;
          xbin=newbin->binx;
          ybin=newbin->biny;
          zbin=newbin->binz;
          /* pointer to list of particles in this bin */
          bin1=bin [xbin] [ybin] [zbin];

          /* find the particle in this bin list and store */
          /* its (x,y,z) location and radius */
```

```
          while((bin1!=NULL)&&(found==0)){

          if(bin1 ->partnum==oldlist->partnum){
               x1=bin1->xloc;
               y1=bin1->yloc;
               z1=bin1->zloc;
               r1=bin1->size;
               found=1;
          }

          bin1=bin1->nextpart;
     }
     if(found==0){
          printf("Error in burn routine, particle not found \n");
          exit(1);
     }

     /* Now compare this particle to all others in the same bin */
     bin1=bin [xbin] [ybin] [zbin];
     while((bin1!=NULL)&&(bin1->partnum!=(-1))){

/* if new particle is not current one, check it for propagation */
if(bin1->partnum!=oldlist->partnum){
     x2=bin1->xloc;
     y2=bin1->yloc;
     z2=bin1->zloc;
     r2=bin1->size;

     /* compute interparticle distance to see if */
     /* burn proceeds */
     /* Center to center distance less than the sum */
     /* of the two radii and the two shell thicknesses */
     dist1=SQR(x2-x1)+SQR(y2-y1)+SQR(z2-z1);
     dist2=SQR(r1+r2+2.*shellthick);
     if(dist1<=dist2){

          /* update connection list as necessary */
          conflag=0;
          connew=conlist;
         while((conflag==0)&&(connew!=NULL)&&(connew->partnum!=(-1))){
               if(connew->partnum==bin1->partnum){
                    conflag=1;
               }
               connew=connew->nextburn;
          }
          if(conflag==0){
               nconnect+=1;
```

```
                    disttot+=(sqrt(dist1)-r1-r2);
                connew=(struct burnlist *)malloc(sizeof(struct burnlist));
                    connew->partnum=bin1->partnum;
                    connew->nextburn=conlist;
                    conlist=connew;
                }

                if(particle[bin1->partnum]->burnt==0){
                    /* propagate burning to this new particle */
                    nburn+=1;
                    burnid[nburn]=bin1->partnum;
                    vtot+=(4.*PIVAL/3.)*CUB(bin1->size);
                    particle[bin1->partnum]->burnt=1;
                 addlist=(struct burnlist *)malloc(sizeof(struct burnlist));
                    addlist->partnum=bin1->partnum;
                    addlist->nextburn=newlist;
                    newlist=addlist;
                 }
            }
            }  /* if partnum */

            bin1=bin1->nextpart;   /* next particle in this bin */

        } /* while bin1 loop */
      newbin=newbin->nextloc;  /* next bin in which current particle exists */
        } /* while newbin loop */

/* process the old burn front as long as one still exists */
/* and free up first element from old list structure */
            addlist=oldlist;
            oldlist=oldlist->nextburn;
            free(addlist);
            /* free up memory in the connected list */
            while(conlist!=NULL){
                connew=conlist;
                conlist=conlist->nextburn;
                free(connew);
            }

        } /* while oldlist #2 */

/* process the new burn front by calling it the old front */
        oldlist=newlist;

        } /* while oldlist #1 */
        free(newlist);
        /* check bottom for connected path through system */
```

```
          botflag=0;
          iz1=NBIN-1;
          for(ix1=0;ix1<NBIN;ix1++){
          for(iy1=0;iy1<NBIN;iy1++){
               bin1=bin [ix1] [iy1] [iz1];

               while((bin1!=NULL)&&(bin1->partnum!=(-1))){
                    if((particle[bin1->partnum]->burnt==1)&&
                     ((bin1->zloc+bin1->size+shellthick)>=SYSIZE)){
                         botflag=1;
                         /* mark these, so they don't */
                         /* get counted more than once */
                         /* Use a special value of 3 */
                         particle[bin1->partnum]->burnt=3;
                    }
                    bin1=bin1->nextpart;
               }
          }
          }

/* update counters */
          ntop+=nburn;
          ntopc+=nconnect;
          if(botflag==1){
               for(ix1=1;ix1<=nburn;ix1++){
                    particle[burnid[ix1]]->burnt=BACKBONEID;
               }
               vtotc+=vtot;
               nthruc+=nconnect;
               disttotc+=disttot;
               nthru+=nburn;
           }
} /* end of if testv loop */

/* move to the next particle in the current top level bin */
          bincur=bincur->nextpart;

          } /* while bincur loop */

/* move to the next bin on the top surface */
     }
     }

     /* output results */
     vfrac=(float)nthru/(float)ntotal;
     vtop=(float)ntop/(float)ntotal;
     aveneigh=0.0;
```

```
      avetopn=0.0;
      avedist=0.0;
      if(nthruc!=0){avedist=disttotc/(float)nthruc;}
      if(nthru!=0){aveneigh=(float)nthruc/(float)nthru;}
      if(ntop!=0){avetopn=(float)ntopc/(float)ntop;}
      printf("\n Particle percolation assessment results \n");
printf("Number frac. accessible - %f  Number frac. through - %f \n",vtop,vfrac);
printf("Ave. num. of connections for particles on backbone is %f \n",aveneigh);
printf("Ave. number of connections for all particles burnt is %f \n",avetopn);
      printf("Average connection distance along backbone is %f \n",avedist);
      printf("Volume through - %f \n",vtotc);
}
```

A.3 Listing for volume.c

```
/*****************************************************************/
/*                                                               */
/*   Program: volume.c                                           */
/*   Purpose: To calculate the volumes occupied by the hard cores */
/*            and soft shells in the HCSS 3-D model              */
/*   Programmer: Kenneth A. Snyder and Dale P. Bentz             */
/*               NIST                                            */
/*               Building 226 Room B-350                         */
/*               Gaithersburg, MD  20899-8621                    */
/*               Phone: (301) 975-5865  Fax: (301) 990-6891      */
/*               E-mail: jackal@nist.gov  dale.bentz@nist.gov    */
/*               WWW page: http://titan.cbt.nist.gov/bentz       */
/*                                                               */
/*****************************************************************/
/*
     A grid is superposed onto the volume and the number of points lying
       within hard/soft spheres are counted.

         resolution : number of points subdividing a single bin

     Note: This routine can be easily modified to output a digitized
          representation of the microstructure if needed for further
          analysis
 */
/* resolution is requested resolution for point counting */
/* in points per bin length */
void volume(resolution)
     float resolution;
{
   int ix,iy,iz,     /* bin indices */
       shellcount,  /* loop counter for shell thicknesses */
       point,        /* descriptor of point: 0-neither 2-soft 3-both */
       burnt_point;  /* descriptor of burnt point: 0-not 20-yes */

   long hard_count,  /* number of points within hard cores */
        soft_count,  /* number of points within soft shells */
        total_count, /* total number of points in system */
        burnt_hard_count, /* number of points within burnt hard cores */
        burnt_soft_count, /* number of points within burnt soft shells */
        acc_soft_count;   /* number of points within accessible soft shells */

   float x,y,z,      /* location of the points for volume calculation */
         stepsize,   /* distance between points */
         binwidth,   /* width of bin */
         shell[2],     /* shell thicknesses (0.0,shellthick) */
         x_lo,x_hi,
```

```
      y_lo,y_hi,
      z_lo,z_hi,  /* limits of bin */
      radius,     /* temp. variable for radius^2 */
      dist;       /* square of distance from point to sphere */

struct partlist *new_bin;  /* linked list of bin information */

/* INITIALIZE VARIABLES */
shell[0] = 0.0;
shell[1] = shellthick;
hard_count = 0;
soft_count = 0;
total_count = 0;
burnt_hard_count=0;
burnt_soft_count=0;
acc_soft_count=0;
binwidth = SYSIZE/(float)NBIN;
stepsize = binwidth/resolution;

   /***  loop through all bins ***/
   /***  and all points in bins ***/
   /***  Interdisperse these two loops ***/
   /***  to allow for easy output of digitized ***/
   /***  microstructure if user would like ***/
for(iz=0;iz<NBIN;iz++) {
     z_lo = binwidth*(float)iz;
     z_hi = z_lo + binwidth-stepsize/2.;
 for(z=z_lo;z<z_hi;z+=stepsize) {
for(iy=0;iy<NBIN;iy++) {
     y_lo = binwidth*(float)iy;
     y_hi = y_lo + binwidth-stepsize/2.;
 for(y=y_lo;y<y_hi;y+=stepsize) {
for(ix=0;ix<NBIN;ix++) {
     x_lo = binwidth*(float)ix;
     x_hi = x_lo + binwidth-stepsize/2.;
 for(x=x_lo;x<x_hi;x+=stepsize) {

 point = 0;
 burnt_point=0;
 total_count ++;    /*** count total number of points ***/
 new_bin = bin[ix][iy][iz];
    /*** loop through all spheres in bin ***/
 while(new_bin->partnum != -1) {
    /* Check for both hard core only and for hard core with soft shell */
    for(shellcount=0;shellcount<2;shellcount++) {
      radius = SQR( new_bin->size+shell[shellcount] );
      dist = SQR(x-new_bin->xloc)+SQR(y-new_bin->yloc)+SQR(z-new_bin->zloc);
```

```
        if(dist<=radius) {
          point |= (shellcount+1);
          burnt_point|=particle[new_bin->partnum]->burnt;
        }
      }
    new_bin = new_bin->nextpart;
  } /* while() */

  if(point>0) {
     if(point>2){
        hard_count ++;   /*** inside both ***/
        if(burnt_point>=BACKBONEID) burnt_hard_count++;
        }
     else     {
        soft_count ++;   /*** only inside soft sphere ***/
        if(burnt_point>=BACKBONEID) burnt_soft_count++;
        if(burnt_point>0) acc_soft_count++;
        }
  }

  /* If user desired to output digitized microstructure */
  /* could simply print value of point to a file here */
  /* 0 - bulk matrix */
  /* 2 - soft shell */
  /* 3 - hard core */

  }}} /* for(points in bin) */
 }}}  /* and(NBINS,NBINS,NBINS) */

 /* Output results of systematic point sampling */
 printf("\n Volume assessment results \n");
 printf("Total points(calculated): %10.0f\n",CUB(resolution)*CUB((float)NBIN));
 printf("Total points(actual)    : %10ld \n",total_count);
 printf("Hard count: %ld\n",hard_count);
 printf("Soft count: %ld\n",soft_count);
 printf("Burnt Hard Count: %ld \n",burnt_hard_count);
 printf("Burnt Soft Count: %ld \n",burnt_soft_count);
 printf("Accessible Soft Count: %ld \n",acc_soft_count);
}
```

A.4 Listing for concrete.c

```
/*************************************************************************/
/*                                                                       */
/*   Program: concrete.c                                                 */
/*   Purpose: To simulate 3-D microstructure of concrete using a         */
/*            hard core - soft shell model                               */
/*   Programmer: Dale P. Bentz                                           */
/*               NIST                                                    */
/*               Building 226 Room B-350                                 */
/*               Gaithersburg, MD  20899-8621                            */
/*               Phone: (301) 975-5865  Fax: (301) 990-6891              */
/*               E-mail: dale.bentz@nist.gov                             */
/*               WWW page: http://titan.cbt.nist.gov/bentz               */
/*                                                                       */
/*************************************************************************/
#include <stdio.h>
#include <math.h>
#include <malloc.h>

#define NPART 500000 /* maximum number of particles */
#define NBIN 30         /* number of bins per dimension */
#define SYSIZE 100.          /* system size per dimension */
#define SQR(a) ((a)*(a))        /* definition for square function */
#define CUB(a) ((a)*(a)*(a))    /* definition of cube function */
#define BACKBONEID 20 /* burnt id for backbone particles */
#define PIVAL 3.1415926         /* definition of pi */
#define PASTE 0                 /* phase ID=0 for bulk matrix (paste) */
#define IZ 1                    /* phase ID=1 for soft shell (intefacial zone) */
#define AGG 2                   /* phase ID=2 for hard core (aggregate) */
#define NUMANTS 10000           /* Number of ants to use in random walk */
#define REALSIZE 30.0           /* real system size in user units (e.g. 30.0 mm) */

/* structure for linked lists of bins for each particle */
struct binlist {
     int binx,biny,binz;       /* Bin identifiers in 3-D system */
     struct binlist *nextloc;  /* Pointer to next bin */
     };

/* structure for each particle containing bin list and burnt flag */
struct aggchar {
     struct binlist *binloc;     /* List of all bins for this particle */
     int burnt;                  /* Burn flag indicating percolation */
     };

/* structure for each bin consisting of a linked list of particles */
/* in that particular bin */
struct partlist {
```

```
      long int partnum;           /* Particle identifier */
      float size;                 /* Particle radius */
      float xloc;                 /* Particle centroid x */
      float yloc;                 /* Particle centroid y */
      float zloc;                 /* Particle centroid z */
      struct partlist *nextpart;  /* Pointer to next particle in bin */
      };

/* structure for diffusing species */
struct walker{
      float x0,y0,z0;             /* Original location */
      float xnew,ynew,znew;       /* Current location in periodic coordinates */
      float xreal,yreal,zreal;    /* Current location in real coordinates */
      float time;                 /* Time elapsed during walk */
      int phasein;                /* Current phase walker is in */
      int active;                 /* Indicates if ant is currently diffusing */
      };


/* Global variables */
struct aggchar *particle[NPART];            /* system particle list */
struct partlist *bin [NBIN] [NBIN] [NBIN];  /* 3-D system of bins */
float shellthick,                           /* shell thickness */
      alpha,beta,gampar;                    /* Weibull dist. parameters */
double ctobin,                              /* NBIN/SYSIZE */
       cfrombin;                            /* 1./NBIN */
static float randrad[NPART];                /* Array of random radii */
int *seed;                                  /* Random number seed */


/* Random number generator ran1 from Computer 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

/* idum is seed value and should be global */
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

/* Include the other c programs needed for the model */
#include "volume.c"          /* point sampling program */
#include "burnt.c"           /* Percolation assessment program */
#include "randnum.c"         /* Random or sieve radii generation */

/* routine to generate a new particle radius according to a Weibull */
/* distribution */
/* See Law and Kelton, Simulation Modelling and Analysis, McGraw-Hill, 1982 */
/* ncur is ID of particle being generated */
/* returns the radius generated */
float radgen(ncur)
     long int ncur;
{
```

```
     float xsubi,              /* Random radii value */
           radback;            /* Corresponding Weibull value */

     /* largest particles first */
     xsubi=randrad[ncur-1];
     radback=beta*pow((-(log(xsubi))),(1.0/alpha))+gampar;
     return(radback);
}

/* routine to initialize data structures (particle and bin lists) */
/* npart is number of particles in system (at start) */
void init(npart)
     long int npart;
{
     /* Loop index variables */
     long int i;
     int ix,iy,iz;

     /* initialize particle list structure */
     for(i=1;i<=npart;i++){
          particle[i]=(struct aggchar *)malloc(sizeof(struct aggchar));
          particle[i]->binloc=NULL;
     }

     /* initialize bin list structure */
     for(ix=0;ix<NBIN;ix++){
     for(iy=0;iy<NBIN;iy++){
     for(iz=0;iz<NBIN;iz++){

          bin [ix] [iy] [iz]=(struct partlist *)malloc(sizeof(struct partlist));
          bin [ix] [iy] [iz] ->partnum=(-1);
          bin [ix] [iy] [iz] ->nextpart=NULL;
     }
     }
     }
}

/* routine to dump complete view of system */
/* Not called during system execution but useful for debugging purposes */
/* ndump is number of particles presently in system */
void dumpsys(ndump)
     long int ndump;
{
     int itest,ix,iy,iz;
     long int i,ipart;
     struct binlist *newpart;          /* Pointer to bin list */
     struct partlist *newbin;          /* Pointer to particle list */
```

```
        /* dump particle list */
        for(i=1;i<=ndump;i++){
             printf("\n Particle - %ld  Bins - ",i);
             newpart=particle[i]->binloc;
             while(newpart!=NULL){
                  printf("%d %d %d",newpart->binx,newpart->biny,newpart->binz);
                  newpart=newpart->nextloc;
             }
        }
        printf("\n");

        /* dump bin information */
        for(iz=0;iz<NBIN;iz++){
        for(iy=0;iy<NBIN;iy++){
        for(ix=0;ix<NBIN;ix++){

             newbin=bin [ix] [iy] [iz];
             itest=0;
             ipart=0;
             /* count all particles in this bin */
             while((newbin!=NULL)&&(newbin->partnum!=(-1))){
                  if(itest==0){
                       itest=1;
                       printf("Particles in bin %d %d %d \n",ix,iy,iz);
                  }
                  ipart+=1;
                  printf("%ld at %f %f %f with radius %f \n",newbin->partnum,
                     newbin->xloc,newbin->yloc,newbin->zloc,newbin->size);
                  newbin=newbin->nextpart;
             }
             printf("%ld \n",ipart);
        }
        }
        }
   }

   /* routine to place the particles in 3-D system (SYSIZE=100 units per side) */
   /* Particles are placed with periodic boundaries in x and y directions */
   /* but routine uses specialized periodic placement in z-direction such */
   /* that two separate particles are created and stored to facilitate the */
   /* burning process */
   /* returns number of particles successfully replaced */
   /* ntopl is number of particle to place */
   /* partdist indicates type of particles (monosize, Weibull, sieve) */
   /* radin is radius for monosize particles */
   int park(ntopl,partdist,radin)
        long int ntopl;
```

```
     int partdist;
     float radin;
{
     int ix,iy,iz,ix1,iy1,iz1;     /* Bin identifiers */
     int flag,                     /* Particle overlap flag */
         izold,idef;               /* For specialized z placement */
     /* Particle coordinates and radii */
     float xtry,ytry,ztry,xnew,ynew,znew,rad1,rad2;
     /* Limits of particle expanse */
     float xmax,xmin,ymax,ymin,zmin,zmax;
     float dist;                   /* Distance overlap check variable */
     float pix4,                   /* 4 pi */
           radall;                 /* Total particle radius (with shell) */
     /* Surface area and shell volume for system */
     double surf_area,shell_vol;
     double rxf;                  /* Test random number after reseeding */
     long int count,              /* Number of tries for a particle */
              i,                  /* Loop index variable */
              partno;             /* Counter for no. of particles placed */
     int tseed;                   /* Seed for reinitialization of ran1 */
     struct partlist *bincheck;   /* Pointer to a single bin being checked */
     struct aggchar *partchk,*partnew;/* Pointers to particles being compared */
     struct binlist *binnew;      /* Pointer to a new bin for a particle */

     /* Initialize surface area and shell volume counts */
     surf_area=0.0;
     shell_vol=0.0;

     /* Initialize random number reseed value */
     tseed=0;
     pix4=4.*PIVAL;
     partno=0; /* counter of particles placed */
     /* place each new particle in turn, being sure that it */
     /* doesn't overlap any existing particles */
     for(i=1;i<=ntopl;i++){
          partno+=1;
          /* Notify user of progress every 10000 particles */
          if((partno%10000)==0){
               printf("Placing particle %ld \n",partno);
               fflush(stdout);
          }
          /* Flag=1 indicates that particle is not yet placed */
          flag=1;
          count=0;    /* Number of tries for this particle */
          /* Determine radius of particle to be placed */
          /* Monosize particles */
          if(partdist==0){
```

```
     rad1=radin;
}
/* Weibull distribution */
if(partdist==1){
     rad1=radgen(i);
}
/* Sieve analysis of PSD */
if(partdist==2){
     rad1=randrad[i];
}
/* Try 5000000 random attempts before giving up */
while((flag==1)&&(count<=500000)){
     flag=0;
     count+=1;
     /* Reinitialize random number seed after 50000 */
     /* failed attempts */
     if((count%50000)==0){
          printf("Calling ran to reinitialize \n");
          tseed-=1;
          *seed=tseed;
          rxf=ran1(seed);
     }
     /* Generate a random location for this particle */
     /* system scale is set at 0 to SYSIZE=100. */
     xtry=SYSIZE*ran1(seed);
     ytry=SYSIZE*ran1(seed);
     ztry=SYSIZE*ran1(seed);
     /* Compute limits of particle expanse */
     xmax=xtry+rad1;
     xmin=xtry-rad1;
     ymax=ytry+rad1;
     ymin=ytry-rad1;
     zmax=ztry+rad1;
     zmin=ztry-rad1;

/* Check all particles in all bins which this particle may contact */
/* for possible overlaps */
for(ix=floor(xmin*ctobin);((ix<=xmax*ctobin)&&(flag==0));ix++){
for(iy=floor(ymin*ctobin);((iy<=ymax*ctobin)&&(flag==0));iy++){
for(iz=floor(zmin*ctobin);((iz<=zmax*ctobin)&&(flag==0));iz++){

     /* conversion for periodic boundaries */
     ix1=(NBIN+ix)%NBIN;
     iy1=(NBIN+iy)%NBIN;
     iz1=(NBIN+iz)%NBIN;

     /* bincheck is pointer to a single bin */
```

```
                bincheck=bin [ix1] [iy1] [iz1];

                /* process all particles in this bin list until an overlap */
                /* is found or all particle evaluated */
                while((flag==0)&&(bincheck!=NULL)&&(bincheck->partnum!=(-1))){

                /* be sure to correct new locations for periodic boundaries */
                xnew=bincheck->xloc+SYSIZE*floor((float)ix*cfrombin);
                ynew=bincheck->yloc+SYSIZE*floor((float)iy*cfrombin);
                znew=bincheck->zloc+SYSIZE*floor((float)iz*cfrombin);
                rad2=bincheck->size;

                /* check for overlap between the two particles */
                dist=((xnew-xtry)*(xnew-xtry)+(ynew-ytry)*(ynew-ytry)+
                     (znew-ztry)*(znew-ztry));

                if(dist<SQR(rad1+rad2)){
                     flag=1;
                }

                /* proceed to next particle in this bin */
                bincheck=bincheck->nextpart;
           }

      }
      }
      }

/* if the particle fits (flag=0), place it by updating data structures */
      if(flag==0){
           /* If particle ID exceeds number requested due to virtual */
           /* particles, allocate new memory as needed */
           if(partno>ntopl){
               particle[partno]=(struct aggchar *)malloc(sizeof(struct aggchar));
                particle[partno]->binloc=NULL;
           }

           partnew=particle[partno];
           partnew->burnt=0;

           /* Compute limits of particle expanse */
           /* this time including the shell */
           xmax+=shellthick;
           xmin-=shellthick;
           ymax+=shellthick;
           ymin-=shellthick;
           zmax+=shellthick;
```

```
          zmin-=shellthick;

/* update bin structure for all bins in which the particle may reside */
          idef=0;
          for(iz=floor(zmin*ctobin);(iz<=zmax*ctobin);iz++){
          for(ix=floor(xmin*ctobin);(ix<=xmax*ctobin);ix++){
          for(iy=floor(ymin*ctobin);(iy<=ymax*ctobin);iy++){

               /* conversion for periodic boundaries */
               ix1=(NBIN+ix)%NBIN;
               iy1=(NBIN+iy)%NBIN;
               iz1=(NBIN+iz)%NBIN;

               /* specialized placement conditions */
               /* requiring creation of a separate virtual particle */
               if(idef!=0){
               if(((izold<0)&&(iz>=0))||((izold<=(NBIN-1))&&(iz>(NBIN-1)))){
                    partno+=1;
                    if(partno>ntopl){
             particle[partno]=(struct aggchar *)malloc(sizeof(struct aggchar));
                         particle[partno]->binloc=NULL;
                    }
                    partnew=particle[partno];
                    partnew->burnt=0;
               }
          }

          /* attach new data to front of linked lists */
          /* add new bin to bin list for this particle */
          binnew=(struct binlist *)malloc(sizeof(struct binlist));
          binnew->binx=ix1;
          binnew->biny=iy1;
          binnew->binz=iz1;
          binnew->nextloc=partnew->binloc;
          partnew->binloc=binnew;

          /* add new particle to the particle list for this bin */
          bincheck=(struct partlist *)malloc(sizeof(struct partlist));
          bincheck->partnum=partno;
          bincheck->size=rad1;
          /* correct location for periodic boundaries */
          bincheck->xloc=xtry-SYSIZE*floor((float)ix*cfrombin);
          bincheck->yloc=ytry-SYSIZE*floor((float)iy*cfrombin);
          bincheck->zloc=ztry-SYSIZE*floor((float)iz*cfrombin);
          bincheck->nextpart=bin [ix1] [iy1] [iz1];
          bin [ix1] [iy1] [iz1]=bincheck;
```

```
          /* Update z bin variables for specialized placement */
          izold=iz;
          idef=1;
          }}}  /* For all bins */
     }   /* if flag loop */

     }   /* while flag loop */
          /* If can't place particle, exit system */
          if(count>500000){
               printf("Couldn't place particle %d in 500000 tries \n",i);
               exit(1);
          }
          /* Else Update surface area and shell volume counts */
          surf_area+=pix4*SQR(rad1);
          radall=rad1+shellthick;
          shell_vol+=(pix4/3.)*(radall*radall*radall-rad1*rad1*rad1);
     }  /* For all particles */

     /* Output results for surface area and shell volume estimate */
     printf("Surface area measured as %lf \n",surf_area);
     printf("Shell volume estimated (neglecting overlaps) as %lf \n",shell_vol);
     return(partno);
}

/* Routine to move ants via random walks to estimate diffusivity */
/* of composite media (diffusivity of hard cores=0, */
/* where the ants are not allowed to tread) */
/* condiz is ratio of SS diffusivity to bulk diffusivity */
/* dstep is the shoe (step) size for each ant step */
/* maxtake is the maximum number of steps for each ant to take */
/* Results are output after maxtake/4, maxtake/2, and maxtake steps */
void moveants(condiz,dstep,maxtake)
     float condiz,dstep;
     long int maxtake;
{
     FILE *outdiff,*outdif1,*outdif2;    /* Output files for results */
     struct walker *ants[NUMANTS+1];     /* linked list of walkers */
     long int iant,istep;                /* Loop indices for ants and steps */
     int ibx,iby,ibz,          /* Location of ant */
         point;                /* Flag indicating ant is trying */
                               /* step into aggregate (no can do) */
     int pold,pnew,            /* Phase identifier for ant local */
         placed;               /* Placement flag for original ant location */
     /* Probabilities for crossing phase boundaries */
     float tiz,ppiz,pizp;
     /* Ant location in 3-D */
     float xtest,ytest,ztest;
     /* Update ant location after a step */
```

```
float xa,ya,za;
float midway;          /* Time update when ant crosses boundary */
float xnonp,ynonp,znonp,   /* Non-periodic location of ant */
      pix2;
/* Radii of particles for comparison purposes */
float dist,radius,tmp1,radius2;
/* Angle variables (cosine and sine) to generate a random direction */
/* for the ant step */
float alpha,gamma,cgamma,cbeta,calpha,sgamma,sbeta,salpha;
struct partlist *test_bin;  /* Pointer to a bin */

/* Initialize variables */
/* Time spent during a step in soft shell is tiz */
/* For explanation of the choice of tiz, pizp, ppiz, and midway */
/* See: Schwartz, L.M., et. al., Journal of Applied Physics, */
/*      Volume 78, 5898-5908, 1995.   */
tiz=1.0/condiz;
pizp=tiz;        /* probability to go from soft shell to bulk */
ppiz=1.0;        /* probability to go from bulk to soft shell */
pix2=2.*PIVAL;
midway=0.5*(1.0+tiz);
/* Initialize data structures for ants */
/* and choose initial positions for ants */
printf("Before initialization in moveants \n");
fflush(stdout);
for(iant=1;iant<=NUMANTS;iant++){
     /* Allocate the needed memory */
     ants[iant]=(struct walker *)malloc(sizeof(struct walker));
     placed=0;
     /* Place the ant either in soft shell or bulk (not in hard core */
     while(placed==0){
          xtest=SYSIZE*ran1(seed);
          ytest=SYSIZE*ran1(seed);
          ztest=SYSIZE*ran1(seed);
          /* Check to be sure point is not in hard core */
          ibx=floor(xtest*ctobin);
          iby=floor(ytest*ctobin);
          ibz=floor(ztest*ctobin);
          test_bin=bin[ibx] [iby] [ibz];
          point=0;
          /* Assume placement is in bulk matrix */
          pold=PASTE;
          while((test_bin->partnum!= -1)&&(point==0)){
               radius=SQR(test_bin->size);
               radius2=SQR(test_bin->size+shellthick);
               dist=SQR(xtest-test_bin->xloc)+SQR(ytest-test_bin->yloc)+
                    SQR(ztest-test_bin->zloc);
```

```
                if(dist<=radius){
                     point=1;
                }
                if(dist<=radius2){
                     /* If in soft shell, update pold */
                     pold=IZ;
                }
                test_bin=test_bin->nextpart;
           }
           /* If not in hard core, place the ant */
           /* and initialize all of its attributes */
           if(point==0){
                placed=1;
                ants[iant]->x0=xtest;
                ants[iant]->xnew=xtest;
                ants[iant]->xreal=xtest;
                ants[iant]->y0=ytest;
                ants[iant]->ynew=ytest;
                ants[iant]->yreal=ytest;
                ants[iant]->z0=ztest;
                ants[iant]->znew=ztest;
                ants[iant]->zreal=ztest;
                ants[iant]->time=0.0;
                ants[iant]->phasein=pold;
                ants[iant]->active=1;
           }
     }
}    /* End of initialization loop */
printf("After initialization in moveants \n");
fflush(stdout);

/* Open files for output of results */
outdiff=fopen("diffuse.out","w");
outdif1=fopen("diffuse.001","w");
outdif2=fopen("diffuse.002","w");

/* Now move each ant in turn until all gone or max steps exceeded */
/* all gone relates to a first passage time problem */
/* and is where the ant->active variable is use */
/* First loop over the number of steps */
for(istep=1;(istep<=maxtake);istep++){
     /* Notify user of progress every 10000 steps */
     if((istep%10000)==0){
          printf("Moving ants step %ld \n",istep);
          fflush(stdout);
     }
     /* Now move each ant in turn */
```

```
for(iant=1;iant<=NUMANTS;iant++){
     pold=ants[iant]->phasein;
     /* Choose random angle on a 3-D sphere */
     alpha=pix2*ran1(seed);
     calpha=cos(alpha);
     salpha=sin(alpha);
     cbeta=(-1.0)+2.0*ran1(seed);
     sbeta=sin(acos(cbeta));
     gamma=pix2*ran1(seed);
     cgamma=cos(gamma);
     sgamma=sin(gamma);
     tmp1=cgamma*cbeta;
     xa=dstep*(tmp1*calpha-salpha*sgamma);
     ya=dstep*(salpha*tmp1+sgamma*calpha);
     za=dstep*(sbeta*cgamma);
     /* Update the ant location */
     xtest=ants[iant]->xnew+xa;
     ytest=ants[iant]->ynew+ya;
     ztest=ants[iant]->znew-za;
     /* Determine the non-periodic location */
     xnonp=ants[iant]->xreal+xa;
     ynonp=ants[iant]->yreal+ya;
     znonp=ants[iant]->zreal-za;
     /* Correct for periodic boundaries */
     if(xtest<0){
          xtest+=SYSIZE;
     }
     else if(xtest>=SYSIZE){
          xtest-=SYSIZE;
     }
     if(ytest<0){
          ytest+=SYSIZE;
     }
     else if(ytest>=SYSIZE){
          ytest-=SYSIZE;
     }
     if(ztest<0){
          ztest+=SYSIZE;
     }
     else if(ztest>=SYSIZE){
          ztest-=SYSIZE;
     }
     /* Determine phase ID of new point */
     /* 1st determine the bin it is located in */
     ibx=floor(xtest*ctobin);
     iby=floor(ytest*ctobin);
     ibz=floor(ztest*ctobin);
```

```
if(ibx==NBIN){ibx=NBIN-1;}
if(iby==NBIN){iby=NBIN-1;}
if(ibz==NBIN){ibz=NBIN-1;}
test_bin=bin[ibx] [iby] [ibz];
pnew=PASTE;
/* Continue until hard core encountered */
/* or all particles in bin are evaluated */
while((test_bin->partnum!= -1)&&(pnew!=AGG)){
     radius=SQR(test_bin->size);
     radius2=SQR(test_bin->size+shellthick);
     dist=SQR(xtest-test_bin->xloc)+SQR(ytest-test_bin->yloc)+
          SQR(ztest-test_bin->zloc);
     if(dist<=radius2){
          pnew=IZ;
     }
     if(dist<=radius){
          pnew=AGG;
     }
     test_bin=test_bin->nextpart;
}
if(pnew!=AGG){
     /* Move from paste */
     if(pold==PASTE){
          /* to paste */
          if(pnew==PASTE){
               ants[iant]->time+=1.0;
               ants[iant]->xnew=xtest;
               ants[iant]->ynew=ytest;
               ants[iant]->znew=ztest;
               ants[iant]->xreal=xnonp;
               ants[iant]->yreal=ynonp;
               ants[iant]->zreal=znonp;
               ants[iant]->phasein=pnew;
          }
          /* to IZ */
          else{
               if(ran1(seed)<=ppiz){
                    ants[iant]->time+=midway;
                    ants[iant]->xnew=xtest;
                    ants[iant]->ynew=ytest;
                    ants[iant]->znew=ztest;
                    ants[iant]->xreal=xnonp;
                    ants[iant]->yreal=ynonp;
                    ants[iant]->zreal=znonp;
                    ants[iant]->phasein=pnew;
               }
               /* staying in paste so inc. time counter by 1 */
```

```
                    else{
                         ants[iant]->time+=1.0;
                    }
               }
          }
          /* Move from IZ */
          else{
               /* to paste */
               if(pnew==PASTE){
                    /* biased probability for allowing move */
                    if(ran1(seed)<=pizp){
                         ants[iant]->time+=midway;
                         ants[iant]->xnew=xtest;
                         ants[iant]->ynew=ytest;
                         ants[iant]->znew=ztest;
                         ants[iant]->xreal=xnonp;
                         ants[iant]->yreal=ynonp;
                         ants[iant]->zreal=znonp;
                         ants[iant]->phasein=pnew;
                    }
                    /* staying in IZ so inc. time counter by tiz */
                    else{
                         ants[iant]->time+=tiz;
                    }
               }
               /* to IZ */
               else{
                    ants[iant]->xnew=xtest;
                    ants[iant]->ynew=ytest;
                    ants[iant]->znew=ztest;
                    ants[iant]->xreal=xnonp;
                    ants[iant]->yreal=ynonp;
                    ants[iant]->zreal=znonp;
                    ants[iant]->phasein=pnew;
                    ants[iant]->time+=tiz;
               }
          }
     }
     /* Step into aggregate not allowed */
     /* but still must advance timer */
     /* Note- step into AGG only possible from IZ */
     /* if step size is less than IZ thickness */
     else{
          ants[iant]->time+=tiz;
     }
     /* If needed, calculate distance travelled */
     /* and output distance and time travelled to file */
```

```
                if((istep==maxtake)){
                     dist=SQR(ants[iant]->xreal-ants[iant]->x0);
                     dist+=SQR(ants[iant]->yreal-ants[iant]->y0);
                     dist+=SQR(ants[iant]->zreal-ants[iant]->z0);
                     fprintf(outdiff,"%f %f\n",dist,ants[iant]->time);
                     ants[iant]->active=0;
                }
                if((istep==(maxtake/2))){
                     dist=SQR(ants[iant]->xreal-ants[iant]->x0);
                     dist+=SQR(ants[iant]->yreal-ants[iant]->y0);
                     dist+=SQR(ants[iant]->zreal-ants[iant]->z0);
                     fprintf(outdif1,"%f %f\n",dist,ants[iant]->time);
                }
                if((istep==(maxtake/4))){
                     dist=SQR(ants[iant]->xreal-ants[iant]->x0);
                     dist+=SQR(ants[iant]->yreal-ants[iant]->y0);
                     dist+=SQR(ants[iant]->zreal-ants[iant]->z0);
                     fprintf(outdif2,"%f %f\n",dist,ants[iant]->time);
                }
           } /* End of iant loop */
           /* Close files as appropriate */
           if(istep==(maxtake/4)){
                fclose(outdif2);
           }
           if(istep==(maxtake/2)){
                fclose(outdif1);
           }
      }    /* End of istep loop */
      fclose(outdiff);
}

main() {
      int nseed,                    /* Random number seed */
          partch;                   /* Particle type selection */
                                    /*    0= monosize */
                                    /*    1= Weibull distribution */
                                    /*    2= Sieve classes */
      long int ntodo,               /* Number of particles requested */
               ndone;               /* Number of particles placed */
      float rsize,                  /* Particle radius for monosize particles */
            resolution,             /* resolution for volume point sampling */
            rstep,                  /* step size for ants */
           diz;                     /* Ratio of SS diffusivity to HC cond. */
      double t1;
      long int maxsteps;            /* Number of steps for ants to take */

      rsize=0.0;
```

```
ctobin=(float)NBIN/(float)SYSIZE;
cfrombin=1./(float)NBIN;

do{
     printf("Enter random number seed value (negative integer)\n");
     scanf("%d",&nseed);
     printf("Random number seed is %d \n",nseed);
} while (nseed>=0);
seed=&nseed;
t1=ran1(seed);
do{
     printf("Enter number of particles to place \n");
     scanf("%ld",&ntodo);
     printf("Particles requested = %ld \n",ntodo);
} while (ntodo<0);

do{
     printf("Do you wish 0) monosize, 1) polysize (Weibull dist.) \n");
     printf("particles or 2) particles based on sieve sizes \n");
     scanf("%d",&partch);
} while ((partch<0)||(partch>2));
if(partch==0){
     printf("Enter size of particles to place \n");
     scanf("%f",&rsize);
     printf("Particle radius = %f \n",rsize);
}
else if(partch==1){
     printf("Enter alpha, beta, and gamma for a Weibull distribution \n");
     printf("of particle sizes \n");
     printf("Please separate entries by a single space \n");
     scanf("%f %f %f",&alpha,&beta,&gampar);
     printf("alpha= %f  beta= %f  gamma= %f \n",alpha,beta,gampar);
     /* Generate a set of uniform random radii between 0 and 1 to be */
     /* transformed to Weibull values later */
     genrand(ntodo);
}
else{
     gensieve(ntodo);
}

printf("Enter shell thickness \n");
scanf("%f",&shellthick);
printf("Shell thickness = %f \n",shellthick);
printf("Enter resolution for volume calc. (points per bin length) \n");
scanf("%f",&resolution);
printf("Resolution set at %f \n",resolution);
printf("Enter relative diffusivity in soft shells (interfacial zones) \n");
```

```
    scanf("%f",&diz);
    printf("Relative diffusivity in soft shells is %f \n",diz);
    printf("Enter ant step size in system units \n");
    scanf("%f",&rstep);
    printf("Step size set at %f \n",rstep);
    printf("Enter maximum number of random steps for each ant to take \n");
    scanf("%ld",&maxsteps);
    printf("Max steps to take set at %ld \n",maxsteps);

    /* Initialize system variables */
    init(ntodo);
    printf("After init routine \n");

    /* Place the particles in the 3-D system */
    ndone=park(ntodo,partch,rsize);
    printf("After park routine with %ld particles \n",ndone);
    fflush(stdout);
    /* Call to dump system information usually not activated */
    /* Useful for debugging purposes */
    /* dumpsys(ndone);  */

    /* Check for percolation of the soft shells */
    burn(ndone);
    fflush(stdout);
    /* Sample 3-D volume to determine phase volume fractions */
    volume(resolution);
    fflush(stdout);
    /* Place and move the ants to calculate diffusivity */
    moveants(diz,rstep,maxsteps);
}
```

B Example datafiles for program execution

B.1 Example input datafile for use with concrete.c

```
-30647      random number seed
68600       number of particles to place
2           particles based on sieve classification
0.10        soft shell thickness in system units
3.0         resolution (sample points per bin per direction) for volume
2.0738      ratio of soft shell to bulk diffusivity
0.03        random walker step size in system units
60000       number of random steps to take
```

B.2 Example sieve classification file- sieve.dat

```
8                          number of sieve classes
12.7 19.05 0.00000667      mimimum diameter, maximum diameter, number fraction
9.525 12.7 0.0000996         for each sieve class
4.75 9.525 0.000255          (diameters are in real physical units, e.g., mm)
2.36 4.75  0.000475
1.18 2.36  0.007725
0.6 1.18   0.051209
0.3 0.6    0.313411
0.15 0.3   0.626821
```

B.3 Example output file- concrun.out

```
Enter random number seed value (negative integer)
Random number seed is -30647
Enter number of particles to place
Particles requested = 68600
Do you wish 0) monosize, 1) polysize (Weibull dist.)
particles or 2) particles based on sieve sizes
Seive 1 - Vlow= 1072.530762 Vhigh= 3619.791016 frac= 0.000007
For particle size 1072.530762 no. of particles is 0
Seive 2 - Vlow= 452.473877 Vhigh= 1072.530762 frac= 0.000100
For particle size 452.473877 no. of particles is 7
Seive 3 - Vlow= 56.115063 Vhigh= 452.473877 frac= 0.000255
For particle size 56.115063 no. of particles is 17
Seive 4 - Vlow= 6.882315 Vhigh= 56.115063 frac= 0.000475
For particle size 6.882315 no. of particles is 33
Seive 5 - Vlow= 0.860289 Vhigh= 6.882315 frac= 0.007725
For particle size 0.860289 no. of particles is 530
Seive 6 - Vlow= 0.113097 Vhigh= 0.860289 frac= 0.051209
For particle size 0.113097 no. of particles is 3513
Seive 7 - Vlow= 0.014137 Vhigh= 0.113097 frac= 0.313411
For particle size 0.014137 no. of particles is 21500
Seive 8 - Vlow= 0.001767 Vhigh= 0.014137 frac= 0.626821
```

```
For particle size 0.001767 no. of particles is 43000
Volume and surface area are 598934.683571 and 535529.847503
Enter shell thickness
Shell thickness = 0.100000
Enter resolution for volume calc. (points per bin length)
Resolution set at 3.000000
Enter relative diffusivity in soft shells (interfacial zones)
Relative diffusivity in soft shells is 2.073800
Enter ant step size in system units
Step size set at 0.030000
Enter maximum number of random steps for each ant to take
Max steps to take set at 60000
After init routine
Placing particle 10000
Placing particle 20000
Placing particle 30000
Placing particle 40000
Placing particle 50000
Placing particle 60000
Surface area measured as 535529.830899
Shell volume estimated (neglecting overlaps) as 59162.435922
After park routine with 69678 particles

 Particle percolation assessment results
Number frac. accessible - 0.384368  Number frac. through - 0.359109
Ave. num. of connections for particles on backbone is 2.157062
Ave. number of connections for all particles burnt is 2.109103
Average connection distance along backbone is 0.100818
Volume through - 623738.000000

 Volume assessment results
Total points(calculated):     729000
Total points(actual)    :     729000
Hard count: 436621
Soft count: 42697
Burnt Hard Count: 378465
Burnt Soft Count: 23953
Accessible Soft Count: 25040
Before initialization in moveants
After initialization in moveants
Moving ants step 10000
Moving ants step 20000
Moving ants step 30000
Moving ants step 40000
Moving ants step 50000
Moving ants step 60000
```