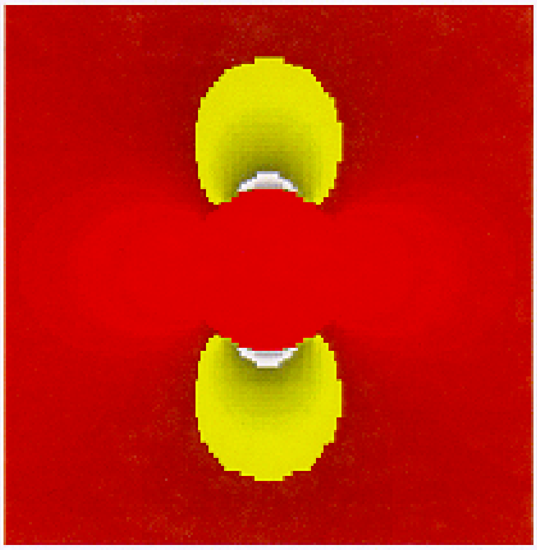
NISTIR 6269

Finite Element and Finite Difference Programs for Computing the Linear Electric and Elastic Properties of Digital Images of Random Materials

Edward J. Garboczi

Building and Fire Research Laboratory
Gaithersburg, Maryland 20899



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

NISTIR 6269

Finite Element and Finite Difference Programs for Computing the Linear Electric and Elastic Properties of Digital Images of Random Materials

Edward J. Garboczi

December 1998
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 Kammer, *Director*

ABSTRACT

This manual has been prepared to show some of the theory behind, and the practical details for using, various finite element and finite difference computer programs that have been developed for computing the effective linear properties of random materials whose microstructure has been stored in a 2-D or 3-D digital image. Thirteen different computer programs are described, including finite element conductivity and elastic programs, finite difference programs for D.C. and A.C. conductivity, finite element elastic programs that include thermal strains (eigenstrains), and three auxiliary programs for Gaussian quadrature and phase percolation. All the programs are written in FORTRAN 77, and operate on an arbitrary digital image that is read from a file. Arbitrary symmetric conductivity tensors and arbitrary elastic moduli tensors can be used in the finite element programs. In the finite difference programs, the conductivity tensors must be diagonal. Only linear elastic and linear electrical conductivity problems are considered. The programs can of course be extended to other problems that have a similar mathematical basis.

Keywords: Building technology, computer modelling, concrete, electrical conductivity, eigenstrain, elastic moduli, finite difference, finite element, linear properties, microstructure, random materials, thermal elasticity, transport properties.

Cover picture: Showing, for a horizontal applied field, the horizontal currents around a circular inclusion of conductivity 10, embedded in a matrix of conductivity one, computed using a finite difference program. The magnitude of the currents go from red = high to black = low, according to the associated color bar.

BLANK PAGE

Contents

List of Figures

List of Tables

1 Introduction

The effective properties of composites have been studied analytically for a long time, back to the days of Maxwell, who first solved for the effect that a single spherical inclusion, with a different conductivity from the matrix, has on the overall or effective conductivity [1]. Random materials like concrete, powder compacts, etc. are also composites, random agglomerations of different property material at various length scales. Because of their randomness, their properties cannot be computed analytically, but instead require numerical computation. To compute the effective properties of such materials requires knowledge of the microstructure. The only practical way such information is obtained is either through microscopy, x-ray microtomography [2, 3], or models [4]. Whatever the method, the microstructural information is almost always in 2-D or 3-D digital image form, collections of discrete square or cubic pixels in which each pixel can, in principle, be a different phase of the material. Hence the need to have computer programs that are specialized to work on digital images.

This manual describes the theory behind, and practical instructions for using, various finite element and finite difference programs specialized to operate on two and three dimensional digital images of materials. The digital images can be completely general, representing any material with any number of distinct phases. The finite element programs are set up assuming that each phase is characterized by an arbitrary symmetry conductivity tensor, elastic moduli tensor, and eigenstrain or thermal strain tensor. The finite difference electrical programs are set up for a conductivity tensor that is diagonal, but which can have different terms along the diagonal for the x, y, and z directions. The digitalimages do not have to have the same dimensions in each direction.

The three basic problems considered are the following: 1) the effective conductivity of a material composed of different conducting and non-conducting phases, either at zero frequency (D.C.) using either a finite element or a finite difference program, or at finite frequency (A.C.), using a finite difference program, 2) the effective elastic moduli of a material composed of phases with different elastic moduli tensors, and 3) the effective thermal strain/eigenstrain of a multi-phase material with different elastic moduli and eigenstrain in each phase. The material digital image is assumed to have periodic boundary conditions (see Section 6 for how to remove these conditions).

The digital images of the microstructures analyzed can be generated within the programs themselves, or can be obtained elsewhere and simply read into the property programs. In the programs described in this manual, the digital images are always read into the program. In the discussion in Section 2, about finite element theory, the language is for 3-D. There is a separate section later in this manual (Sec. 4.4), which explains the quite simple relations between 2-D and 3-D. The 2- D programs in this package were created directly from the 3-D programs, and so are nearly identical in structure to the 3-D versions.

There are 10 main programs described in this manual, and available for the user. They include D.C. finite element electrical programs in 2-D and 3-D (ELECFEM2D.F, ELECFEM3D.F), finite element elastic programs in 2-D and 3-D (ELAS2D.F, ELAS3D.F), finite element elastic programs that incorporate thermal strains (eigenstrains) in 2-D and 3-D (THERMAL2D.F, THERMAL3D.F), and finite difference electrical programs for a.c. or d.c. problems in 2-D or 3-D (AC2D.F, AC3D.F, DC2D.F, DC3D.F). In what follows, any mathematical variable will be in italics. All program and subroutine names will be in capitals. Also, a variable with two subscripts, say w_{mn}, may be written as $w(m, n)$, which is the way it appears in the programs. As much as possible, variable and subroutine names are the same from program to program. The notation and structure of the conjugate gradient routine (in subroutine DEMBX) has been made to follow that used in Numerical Recipes, chapter 10 [5]. Also, in the following, the term "vector" is used interchangeably for 3-D and 2-D vectors, for arrays that have as many entries as there are variables in the problem, and for quantities that have both magnitude and direction. The meaning will be clear from the context. Enough details are given in this manual so that, hopefully, any user can rewrite and change the programs to adapt them to specific problems, and add new capabilities.

All programs should be run in double precision (8 bytes or 64 bits per real variable). Running the finite difference routines in single precision may give reasonable answers, but double precision is more trustworthy. Running the finite element routines in single precision will not work, except perhaps for very small systems, on the order of 1000 nodes. All the routines are written in simple FORTRAN 77, making use of only a small subset of the language. These routines should therefore be portable across different platforms. Those users who wish to proceed directly to how to run the programs, can skip to Sec. 4.5, where details of how to use the programs are discussed. Another similar package, which is for 2-D only but has a graphical interface, is the OOF system, which can be found at *http://www.ctcms.nist.gov/* (click on "Software").

In addition to these 10 main programs, there are also three auxiliary programs, one for computing Gaussian quadrature weights and points, and two for computing phase percolation in an arbitrary 3-D or 2-D image.

The layout of the remainder of this manual is the following. Section 2 briefly derives and defines the theory behind the finite element programs, electric, elastic, and elastic with thermal or eigenstrains. Section 3 discusses the theory behind the finite difference programs, both A.C. and D.C. The early parts of Section 4 explains the workings of key subroutines of both the finite element and finite difference programs, followed by a discussion of how to use the programs and which variables need to be set by the user. Section 5 then presents the theoretical results of many exact solutions in electrical conductivity and elasticity, accompanied by tests or examples showing how well these programs can reproduce these results. These exact results are helpful in testing progams such as the ones presented in this manual, since they are for non-trivial microstructures and/or choices of individual phase paramters. Section 6 then presents the theory showing how these programs may be modified to solve many other problems of interest, along with some useful examples of program use. Section 7 explains how to make images and histograms from the local field results using any of the programs, and gives some examples, and Section 8 is the reference section. Section 9, the last section, gives computational details, access information for the programs, and includes listings of five of the 10 main programs, and two of the auxiliary programs.

1.1 Acknowledgements

I would like to thank Prof. A.R. Day, of Marquette University, for setting up the first version of the conjugate gradient routine for the finite element problems, Prof. P.M. Duxbury, of Michigan State University, for giving me a version of what eventually was developed into the basic 2-D finite difference electric program, Prof. M.F. Thorpe, for getting me started in algorithms applied to digital images, J.F. Douglas, for suggesting some of the example problems for exact composite results, D.P. Bentz, of the NIST Building Materials Division, who has been a collaborator for almost a decade on digital-image-based microstructure problems, and S.A. Langer, for critically reading this manual and suggesting many changes.

2 Finite element theory

2.1 General aspects and node labelling scheme

The theory on which the finite element programs described in this manual are based is very simple. The essential idea is that a variational principle exists for the linear elastic and linear electrical conductivity problems. For a given microstructure, subject to applied fields or other boundary conditions, the final voltage or elastic displacement distribution is such that the total energy stored in the elastic case, or the total energy dissipated, in the electrical conductivity case, is extremized, such that the gradient of the energy with respect to the variables of the problem (voltage or elastic displacement) is zero. The variable En is used for both of these cases, and is called an energy in all the following text, even though, in the case of electrical conductivity, it is really an energy dissipation per unit time or power. To minimize En, a function of many variables u_m , the various partial derivatives must equal zero,

$$\frac{\partial En}{\partial u_m} = 0 \tag{1}$$

for all values of m. In all the programs, the sum of the squares of all elements of the gradient vector, whose m'th element is just the partial derivative in eq. (1), is determined during the solution or relaxation process. The solution of the problem is considered to be reached when this sum is less than a given small value, so that the condition in eq. (1) is approximately satisfied for all m. This value, denoted *gtest* in all the programs, should be chosen small enough so that the answers obtained, the currents or stresses in the pixels, are no longer changing significantly with further relaxation.

A labelling scheme must be defined for a single element or pixel in order to derive the finite element equations for that pixel. "Pixel" and "element" will be used interchangeably throughout this manual. In the finite element method, each node attached to a corner of a pixel (8 in 3-D, 4 in 2-D) has a separate label within that pixel. Since in the finite element method, the energy is defined within each pixel using only the nodes attached to that pixel, it is important to be able to refer easily to these attached nodes. The (i, j, k) label for a pixel, which gives its position in a three-dimensional lattice, is the same as the (i, j, k) label for the node numbered 1 in the pixel. Table 1 gives this single-pixel labelling scheme, in terms of the Δi, Δj, and Δk positions of the nodes with respect to (i,j,k). Figure 1 shows this labelling scheme graphically, and also defines the coordinate system used. The (i,j,k) axes coincide with the (x,y,z) axes, respectively.

The basic derivations for the electric case and the elastic case will now be reviewed, with the pixel length taken to be unity. Most books on finite elements will have much of this derivation. It is given here in order to clarify the structure of the programs described in this manual. In all the discussion below, r and s run from 1 to 8, and indicate the node of the finite element being considered, while p and q run from 1 to 3, and indicate the Cartesian coordinate ($1 = x$, $2 = y$, $3 = z$) of vector quantities. The convention is used throughout the rest of the manual that if a subscript is repeated, it is assumed to be summed over.

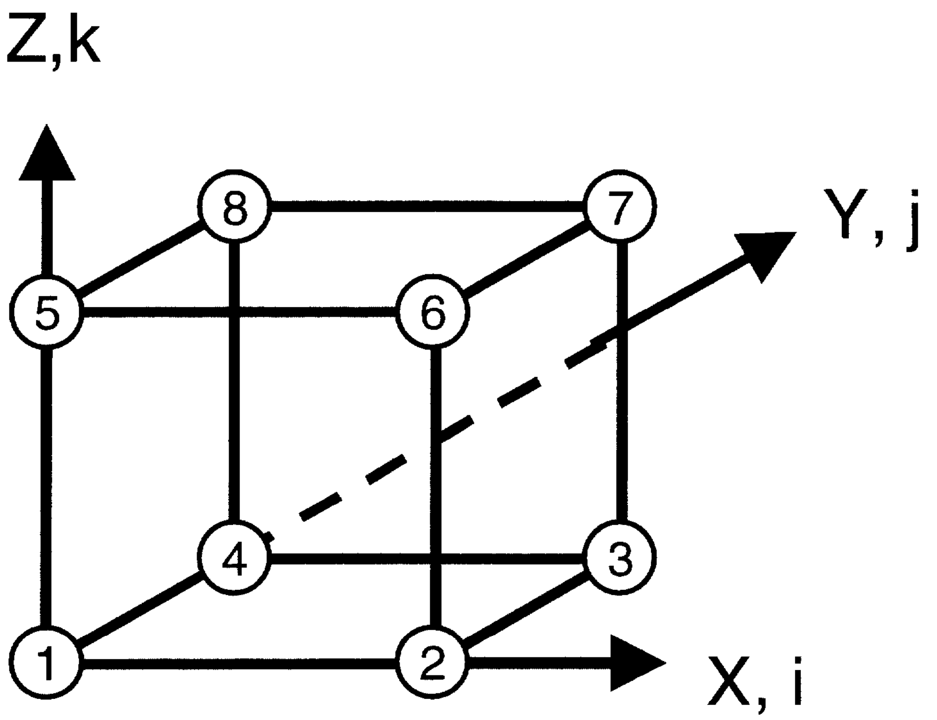


Figure 1: Graphic view of cubic pixel = tri-linear finite element, showing the 1-8 labels of the vertices. The i,j,k axes coincide with the x,y,z axes.

Δi	Δj	Δk	fem label (3-D)	fem label (2-D)
0	0	0	1	1
1	0	0	2	2
1	1	0	3	3
0	1	0	4	4
0	0	1	5	
1	0	1	6	
1	1	1	7	
0	1	1	8	

Table 1: Finite element (fem) labels for within a single pixel labelled (i,j,k). The Δi, etc. values are with respect to the node labelled (i,j,k).

2.2 Electrical conductivity

Variables in programs

u_r = voltage at the r'th node in a pixel
σ_{pq} = conductivity tensor (differs in general from element to element)
$\vec{E} = (E_x, E_y, E_z)$ = external electric field applied to image
$e = (e_x, e_y, e_z)$ = local field at a point (x, y, z) inside a pixel
D_{rs} = stiffness matrix in a pixel
$N_r(x, y, z)$ = shape array for cubic pixel

The purpose of this kind of finite element method is to express the total energy of the system in terms of the voltages only at the nodes, thus effectively discretizing the continuum. There are many ways in which to do this. We choose here to use a linear interpolation method, in what is often called a tri-linear scheme. Essentially we write out the energy of one pixel, which involves an integral over that pixel, and then combine the pixels of the whole system into a global energy functional. The fields inside the pixel are expressed as functions of (x, y, z) via a linear interpolation scheme in terms of the nodal voltages. The integral is carried out, and we end up with an expression for the energy of the pixel that is quadratic in the nodal voltages. The global energy is then also a quadratic functional of the nodal voltages. This expression is minimized with respect to the nodal voltages, using a conjugate gradient scheme. Using the set of nodal voltages found, the interpolation scheme can then be re-used to find the average current, total energy, etc. that can be used to define the effective conductivity and other quantities of interest (see Section 5.1).

Focus on a single pixel for now. Define $V(x, y, z)$ to be the voltage within the given pixel, where $0 < x, y, z < 1$. The origin is taken where the pixel label is 1 (see Table 1 and Fig.

1). $V(x, y, z)$ is determined by linear interpolation of the nodal voltages, such that

$$V(x, y, z) = N_r u_r \tag{2}$$

and $N_r = N_r(x, y, z)$. The actual functional forms of N_r in 3-D are:

$$\begin{aligned}
N_1 &= (1-x)(1-y)(1-z) \\
N_2 &= x(1-y)(1-z) \\
N_3 &= xy(1-z) \\
N_4 &= (1-x)y(1-z) \\
N_5 &= (1-x)(1-y)z \\
N_6 &= x(1-y)z \\
N_7 &= xyz \\
N_8 &= (1-x)yz
\end{aligned}$$

Now the local field in the pixel, $\vec{e}$, is given in terms of derivatives of this local voltage, via

$$e_p(x, y, z) = -\frac{\partial V(x, y, z)}{\partial x_p} \tag{3}$$

In terms of the nodal voltages, this expression for e_p (the p'th component of the local field $\vec{e}$) then becomes

$$e_p(x, y, z) = \frac{-\partial}{\partial x_p}\left[N_r u_r\right] = \left[\frac{-\partial N_r}{\partial x_p}\right] u_r \tag{4}$$

where $-\partial N_r/\partial x_p \equiv n_{pr}$ is a 3×8 matrix linking the 8-vector of nodal voltages to the 3-vector of local fields, and u_r has no explicit x dependence. Table 2 gives the components of this matrix.

r	$-\partial N_r/\partial x \equiv n_{1r}$	$-\partial N_r/\partial y \equiv n_{2r}$	$-\partial N_r/\partial z \equiv n_{3r}$
1	$(1-y)(1-z)$	$(1-x)(1-z)$	$(1-x)(1-y)$
2	$-(1-y)(1-z)$	$x(1-z)$	$x(1-y)$
3	$-y(1-z)$	$-x(1-z)$	xy
4	$y(1-z)$	$-(1-x)(1-z)$	$(1-x)y$
5	$(1-y)z$	$(1-x)z$	$-(1-x)(1-y)$
6	$-(1-y)z$	xz	$-x(1-y)$
7	$-yz$	$-xz$	$-xy$
8	$(1-y)z$	$-(1-x)z$	$-(1-x)y$

Table 2: Formulas for components of $[\partial N_r/\partial x_i] \equiv n_{ir}$ matrix

Now, the total energy dissipated in the pixel is

$$En = \int_0^1 \int_0^1 \int_0^1 dx\, dy\, dz \left[\frac{1}{2} e_p \sigma_{pq} e_q\right] \tag{5}$$

In terms of the nodal voltages, this equation becomes

$$En = \frac{1}{2}\, u_r \, \left[\int_0^1 \int_0^1 \int_0^1 dx\, dy\, dz \left[n_{pr}^T \sigma_{pq} n_{qs}\right]\right] \, u_s \tag{6}$$

The indicated integrations can be easily carried out exactly using Simpson's rule, since they are at most quadratic in x, y, or z, and Simpson's rule is exact for quadratic expressions. These expressions could also be evaluated analytically, and then put into the program. When these programs were developed, it was decided to use Simpson's rule for simplicity, and to avoid doing all the derivations necessary to get the "stiffness" matrices in analytical form. The subroutine FEMAT in the finite element programs carries out this integration. The energy per pixel is then

$$En = \frac{1}{2} u_r D_{rs} u_s \tag{7}$$

where the matrix D_{rs}, an 8×8 matrix in 3-D, is defined by comparing eq. (7) to eq. (6). This matrix is known as a "stiffness" matrix, a term originating in finite element treatments of linear elasticity problems. The above term is added up for all pixels to give the global energy, each pixel using its own value of the conductivity tensor. This sum is then the total energy dissipated of the system under the applied field, which must be minimized with respect to the nodal voltages to find the solution to this discretized version of the original problem.

In order to be able to link up local energy with the global energy, it is important to see how the local numbering scheme for the pixel nodes links up with the global numbering scheme. The digital images are in the form $pix(m)$, where pix is a 2-byte integer vector (defined INTEGER*2 in FORTRAN 77), and m is a one dimensional label. The digital image in 3-D is defined by (i, j, k), where $i = 1, nx$, $j = 1, ny$, and $k = 1, nz$, with the origin of the image being the $(1, 1, 1)$ node. The one dimensional labelling scheme is defined by:

$$m = nx \cdot ny \cdot (k-1) + nx \cdot (j-1) + i \tag{8}$$

The label m refers to the m'th node, and the m'th pixel. The m'th pixel has the m'th node at its corner that is labelled "1" in the $1-8$ local finite element numbering scheme. In 3-D , every node is part of 8 pixels. All the nodes in these pixels will need to be known with respect to the node, since they are connected to it in the global energy via the local stiffness matrices, and so their labels are encoded in the array ib. If the m'th node corresponds to (i, j, k), then the 27 nodes that are part of these 8 pixels are given by the m labels corresponding to adding or subtracting 0 or 1 from the i,j, k labels, so that $ib = ib(m,27)$ has 27 entries for each value of m. Table 3 gives this neighbor labelling scheme. For example, $ib(m,15)$ would be the m label of the node located at $(-1,0,-1)$ with respect to the (i,j,k) position of the m'th node. Table 3 also gives the neighbor labelling scheme for 2-D, where $\Delta k = 0$, since there is no third dimension. Note that in the interior of an image, the m labels of the neighbors would always be simply known anyway, because of their fixed relationship in

Δi	Δj	Δk	3-D neighbor #	2-D neighbor #
0	1	0	1	1
1	1	0	2	2
1	0	0	3	3
1	-1	0	4	4
0	-1	0	5	5
-1	-1	0	6	6
-1	0	0	7	7
-1	1	0	8	8
0	0	0	27	9
0	1	-1	9	
1	1	-1	10	
1	0	-1	11	
1	-1	-1	12	
0	-1	-1	13	
-1	-1	-1	14	
-1	0	-1	15	
-1	1	-1	16	
0	0	-1	25	
0	1	1	17	
1	1	1	18	
1	0	1	19	
1	-1	1	20	
0	-1	1	21	
-1	-1	1	22	
-1	0	1	23	
-1	1	1	24	
0	0	1	26	

Table 3: Neighbor labelling in 2-D and 3-D

fem label (3-D)	neighbor label (3-D)	fem label (2-D)	neighbor label (2-D)
1	27	1	9
2	3	2	3
3	2	3	2
4	1	4	1
5	26		
6	19		
7	18		
8	17		

Table 4: Relation between finite element labelling and neighbor labeling

terms of i, j, and k, but with periodic boundary conditions (discussed below) a "neighbor" could be at the other side of the image. The relationship between the local $(1-8)$ labelling, and the global $(1-27)$ neighbor labelling used in array ib, is given in Table 4.

Periodic boundary conditions mean that if a neighbor is outside the digital image, it is periodically continued on the opposite side of the system. For example, consider the node at $i = 10$, $j = 12$, $k = 20$, in an $nx = ny = nz = 20$ digital image. The m label of this node (pixel) would be 7830, via eq. (8). Neighbor number 18 would be located at $i = 11$, $j = 13$, and $k = 21$. However, $k = 21 > nz$, so the k label is changed to $k - nz = 21 - 20 = 1$. The m label of $i = 11$, $j = 13$, $k = 1$ is $m = 251$, so that $ib(7830, 18) = 251$. The process is similar for the other faces $(i = nx, j = ny)$, the edges ($i = nx$ and $j = ny$, $i = nx$ and $k = nz$, $j = ny$ and $k = nz$) and the corner point $(i = nx, j = ny, k = nz)$.

Before going into the solution technique, one must first consider the boundary conditions of the problem. Equation (7) is positive definite, and so has its minimum value when all the voltages are zero, an uninteresting result. We use periodic boundary conditions to apply an electric field, so that the energy is minimized with respect to an applied field, a more useful case.

Consider a pixel as before, with $i = nx, j < ny, k < nz$, and label m. When we go to evaluate the energy expression above, we find no voltages at the 2,3,6, and 7 nodes, because these would have i labels of $(nx+1)$, which is undefined. We use the voltages from across the system, numbers 1,4,5, and 8 from the pixel with the same j and k label but with $i = 1$, and label $M = m - nx + 1$. However, there is a jump of nx in length from where these voltages are defined. If the applied field $\vec{E} = (E_x, E_y, E_z)$ is present, then on average there is a drop in voltage of $-E_x nx$ between these nodes. Therefore, the voltages to be used in the energy expression for the m'th pixel are: $u_1 = u_1(m)$, $u_2 = u_1(M) - E_x\, nx$, $u_3 = u_4(M) - E_x\, nx$, $u_4 = u_4(m)$, $u_5 = u_5(m)$, $u_6 = u_5(M) - E_x\, nx$, $u_7 = u_8(M) - E_x\, nx$, and $u_8 = u_8(m)$.

We can write this, in general for a pixel at a boundary, as $u_r = U_r + \delta_r$, where U_r is an 8-vector of the voltages that are given by the $ib(m, n)$ labels, and δ_r is an 8-vector that corrects them to what they should be in the pixel in question. The neighbor array ib is designed to pick up the appropriate nodal voltages from across the image when evaluating the energy of pixels on faces, edges, and corner.

Inserting this choice of voltages into the expression for the energy involving the stiffness matrix, eq. (7), we get for the given pixel

$$En = \frac{1}{2}\left[u_r D_{rs} u_s + 2\delta_r D_{rs} u_s + \delta_r D_{rs} \delta_s\right] \tag{9}$$

which gives a term quadratic in the nodal voltages, a term linear in the nodal voltages, and a term constant with respect to the nodal voltages. In the particular case discussed, $i = nx$, $j < ny$, $k < nz$, the components of the 8-vector δ_r are: $\delta_1 = 0$, $\delta_2 = -E_x\, nx$, $\delta_3 = -E_x\, nx$, $\delta_4 = 0$, $\delta_5 = 0$, $\delta_6 = -E_x\, nx$, $\delta_7 = -E_x\, nx$, $\delta_8 = 0$.

We can rewrite eq. (9) as

$$En = \frac{1}{2} u_r D_{rs} u_s + b_r u_r + C \tag{10}$$

where

$$b_s = \delta_r D_{rs} \ , \quad C = \frac{1}{2}\delta_r D_{rs} \delta_s \tag{11}$$

Adding these up over every pixel is done in the subroutine FEMAT, giving a global array b that gives the term in the energy that is linear in the voltages, and a global constant C. It is important to remember that the only contributions to b and C come from pixels having nodes at the unit cell boundaries and having a non-zero stiffness matrix. That makes it easy to make non-periodic boundary conditions by surrounding the system of interest by a layer of insulating material, one pixel thick, which effectively gets rid of the periodic boundary conditions and makes b and C equal to zero (see Sec. 6.2). Table 5 shows the values of the δ_r variable for the various faces and edges of the digital image.

Once the energy equation is set up, all that remains is to find the set of voltages that minimize the electrical energy dissipated. This is done by a conjugate gradient method, similar to that described in Ref. [5]. The subroutine DEMBX contains this routine, and is the same for both electric and elastic problems, aside from some minor labelling differences of indices, due to the fact that voltage is a scalar while elastic displacement is a 3-D vector. (see Section 4.2). No preconditioning of the matrix to be solved is done. It is not clear, since the full Hessian matrix is never stored, whether it would even be possible to carry out pre-conditioning in this case.

In the subroutine ENERGY, the gradient of the energy is computed, as this must become very small in order to solve the problem, which is exactly solved when the gradient is zero. Remember that the gradient is a vector containing all the partial derivatives of the energy with respect to all the nodal voltages. Using the above expression, the gradient of the energy will then be

$$\frac{\partial En}{\partial u_m} = A_{mn} u_n + b_m \tag{12}$$

where now A_{mn} and b_m should be thought of as global quantities, so that all the terms connected to u_m are included. The global matrix A_{mn} is of course built up from the individual D_{rs} matrices of the eight pixels that touch the node labelled m. The matrix A_{mn} is in principle large, but sparse. The way the finite element programs save memory is by

r	i = nx	j = ny	k = nz	i = nx j = ny	i = nx k = nz	j = ny k = nz	i = nx j = ny k = nz
1	0	0	0	0	0	0	0
2	$-E_x\, nx$	0	0	$-E_x\, nx$	$-E_x\, nx$	0	$-E_x\, nx$
3	$-E_x\, nx$	$-E_y\, ny$	0	$-E_x\, nx$ $-E_y\, ny$	$-E_x\, nx$	$-E_y\, ny$	$-E_x\, nx$ $-E_y\, ny$
4	0	$-E_y\, ny$	0	$-E_y\, ny$	0	$-E_y\, ny$	$-E_y\, ny$
5	0	0	$-E_z\, nz$	0	$-E_z\, nz$	$-E_z\, nz$	$-E_z\, nz$
6	$-E_x\, nx$	0	$-E_z\, nz$	$-E_x\, nx$	$-E_x\, nx$ $-E_z\, nz$	$-E_z\, nz$	$-E_x\, nx$ $-E_z\, nz$
7	$-E_x\, nx$	$-E_y\, ny$	$-E_z\, nz$	$-E_x\, nx$ $-E_y\, ny$	$-E_x\, nx$ $-E_z\, nz$	$-E_y\, ny$ $-E_z\, nz$	$-E_x\, nx$ $-E_y\, ny$ $-E_z\, nz$
8	0	$-E_y\, ny$	$-E_z\, nz$	$-E_y\, ny$	$-E_z\, nz$	$-E_y\, ny$ $-E_z\, nz$	$-E_y\, ny$ $-E_z\, nz$

Table 5: Values of δ_r variable for system

computing $A_{mn}\, u_n$ using only the small single-pixel stiffness matrices and the appropriate labelling scheme. This algorithm can be seen in subroutines ENERGY and DEMBX of the finite element programs. The correct D_{rs} term is used in place of the A_{mn} term that would be needed. In essence, the matrix A is re-built every time it is needed, without being stored.

2.3 Elastic moduli

The elasticity problem is set up similarly to the electrical conductivity problem, because the elastic energy stored also obeys a variational principle. The elastic energy stored is given by

$$En = \frac{1}{2} \int d^3r\, \epsilon_{pq}\, C_{pqrs}\, \epsilon_{rs} \tag{13}$$

where the strain tensor ϵ_{pq} and elastic moduli tensor C_{pqrs} are in full tensorial form, p,q,r,s = 1, 2, or 3, and the integral is over the volume of a single pixel. The total energy is obtained by summing over all pixels. Since the strain tensor is symmetric, a simpler notation is usually used, the Voigt notation, where the strain is taken to be a 6-vector containing the six independent strains($\epsilon_{xx}, \epsilon_{yy}, \epsilon_{zz}, \epsilon_{xz}, \epsilon_{yz}, \epsilon_{xy}$), and then C_{pqrs} is written as $C_{\alpha\beta}$, where

$$\epsilon_{pp} = \frac{\partial u_p}{\partial x_p}$$

$$\epsilon_{pq} = \frac{\partial u_p}{\partial x_q} + \frac{\partial u_q}{\partial x_p}$$

We will use α and β exclusively as labels running over the six components of the strain 6-vectors. The energy equation becomes

$$En = \frac{1}{2}\int d^3r\, \epsilon_\alpha\, C_{\alpha\beta}\, \epsilon_\beta \tag{14}$$

The idea of the finite element scheme, as for the electrical case, is to reduce the energy equation to a quadratic form containing the components of the elastic displacement vector defined at the nodes of the pixels. There is an elastic displacement defined at each node, which has three components in 3-D. We denote this by $u(m,3)$. The m and ib integer labelling system is the same as before, but now all real variables have an extra label for the Cartesian component being considered:

u_{mp} = p'th component of displacement at m'th node
$C_{\alpha\beta}$ = elastic moduli tensor (Voigt notation) for a pixel
$\vec{E} = (E_{xx}, E_{yy}, E_{zz}, E_{xz}, E_{yz}, E_{xy})$ = overall elastic strains applied to system
$\vec{\epsilon} = (\epsilon_{xx}, \epsilon_{yy}, \epsilon_{zz}, \epsilon_{xz}, \epsilon_{yz}, \epsilon_{xy})$ = local strains at a point (x,y,z) in a pixel
$D_{rp,sq}$ = stiffness matrix in a pixel
$N_{p,rq}$ = shape matrix for cubic pixel

Each component of the displacement is linearly interpolated across the pixel in exactly the same way as the voltage was done in the electric problem. The p'th component of the three-vector $\vec{u}(x,y,z)$ at a point (x,y,z) in the pixel is then defined as

$$u_p(x,y,z) = N_{p,rq}(x,y,z)u_{rq} \tag{15}$$

where $N_{1,r1} = N_{2,r2} = N_{3,r3} = N_r$ given for the electrical problem, and u_{rq} is the q'th component of the displacement on the r'th node, $r = 1,8$. Clearly, in this structure, we have $N_{1,r2} = N_{1,r3} = N_{2,r1} = N_{2,r3} = N_{3,r1} = N_{3,r2} = 0$. N is a $3 \times (8,3)$ matrix. To construct the 6-vector strain from this, we need to multiply by (or operate with) a matrix of derivatives, L_{pq}, that is 6×3. The components L_{pq} are given in Table 6 below. This results in

$$\epsilon_\alpha(x,y,z) = \left[L_{\alpha p}N_{p,rq}(x,y,z)\right] u_{rq} \tag{16}$$

or

$$\epsilon_\alpha(x,y,z) = S_{\alpha,rq}(x,y,z)u_{rq} \tag{17}$$

where the components of $S_{\alpha,rq}$ can be found in ELAS3D.F, in the subroutine FEMAT, where the matrix $es(n1,n2,n3)$ is equivalent to $S_{\alpha,rq}$. If eq. (17) is integrated over a pixel, it gives the average strain in the pixel, in terms of the nodal displacements. When first multiplied by $C_{\alpha\beta}$ and then integrated, the average stress in the pixel results. When the solution displacements to the problem are obtained, these can be used to compute the average stress and strain in the system, which define the effective quantities and can also give insight into local stress and strain fields around microstructural features.

Substituting into the energy expression above, in the Voigt notation, results in

$$En = \frac{1}{2}\int d^3r \left[S_{\alpha,rp}u_{rp}\right]^T C_{\alpha\beta} \left[S_{\beta,sq}u_{sq}\right] \tag{18}$$

p (component of strain vector)	q = 1 (x)	q = 2 (y)	q = 3 (z)
1	$\partial/\partial x$	0	0
2	0	$\partial/\partial y$	0
3	0	0	$\partial/\partial z$
4	$\partial/\partial z$	0	$\partial/\partial x$
5	0	$\partial/\partial z$	$\partial/\partial y$
6	$\partial/\partial y$	$\partial/\partial x$	0

Table 6: Components of L_{pq}

Grouping the S and C matrices together, and performing the integral over the pixel using the part that has (x, y, z) dependence, results in

$$En = \frac{1}{2} u_{rp}^T D_{rp,sq} u_{sq} \tag{19}$$

where

$$D_{rp,sq} = \int d^3r \, [S_{\alpha,rp}]^T \, C_{\alpha\beta} \, [S_{\beta,sq}] \tag{20}$$

is the stiffness matrix. It is also the same as the dynamical matrix arising in lattice models of elastic phenomena found in theoretical physics, where displacements at nodes are connected by various forces [6]. The first part of subroutine FEMAT computes the stiffness matrix (*dk* in the finite element programs) using Simpson's rule to perform the integration. As in the electrical case, the integration is exact, as there is no term to be integrated that is higher order than quadratic, and Simpson's rule is exact for quadratic functions.

Periodic boundary conditions result in exactly the same structure as in the electrical case, but with now the extra index for the Cartesian coordinates, and a different form for the vector δ_{rp} , expressed in terms of the six independent applied strains. The components of δ_{rp} are given in Table 7. The periodic boundary conditions result in a term linear in the displacements, denoted $b \cdot u$, as well as a constant term C that is quadratic in the applied strains. The gradient of the energy is the same as in the electric case, but with the extra index for the Cartesian coordinates of the displacement. Note that the terms of b, by definition, are linear in the applied strains (see 2.2). This is important for the development of the eigenstrain or thermal strain case, discussed next.

2.4 Thermal strains (eigenstrains)

In the case of thermal strains, sometimes called eigenstrains (terms used interchangeably in this manual) [7], each phase can have a stress-free strain that comes about by thermal or moisture expansion/shrinkage, or other causes. We denote this strain as e_α, where α =1,6 as usual in the Voigt notation. The stress then becomes $\sigma_\alpha = C_{\alpha\beta}(\epsilon_\beta - e_\beta)$, where ϵ_β is the

r,p	i = nx	j = ny	k = nz	i = nx j = ny	i = nx k = nz	j = ny k = nz	i = nx j = ny k = nz
1,1	0	0	0	0	0	0	0
1,2	0	0	0	0	0	0	0
1,3	0	0	0	0	0	0	0
2,1	$E_{xx}\,nx$	0	0	$E_{xx}\,nx$	$E_{xx}\,nx$	0	$E_{xx}\,nx$
2,2	$E_{xy}\,nx$	0	0	$E_{xy}\,nx$	$E_{xy}\,nx$	0	$E_{xy}\,nx$
2,3	$E_{xz}\,nx$	0	0	$E_{xz}\,nx$	$E_{xz}\,nx$	0	$E_{xz}\,nx$
3,1	$E_{xx}\,nx$	$E_{xy}\,ny$	0	$E_{xx}\,nx$	$E_{xx}\,nx$	$E_{xy}\,ny$	$E_{xx}\,nx$ $+E_{xy}\,ny$
3,2	$E_{xy}\,nx$	$E_{yy}\,ny$	0	$E_{xy}\,nx$	$E_{xy}\,nx$	$E_{yy}\,ny$	$E_{xy}\,nx$ $+E_{yy}\,ny$
3,3	$E_{xz}\,nx$	$E_{yz}\,ny$	0	$E_{xz}\,nx$	$E_{xz}\,nx$	$E_{yz}\,ny$	$E_{xz}\,nx$ $+E_{yz}\,ny$
4,1	0	$E_{xy}\,ny$	0	$E_{xy}\,ny$	0	$E_{xy}\,ny$	$E_{xy}\,ny$
4,2	0	$E_{yy}\,ny$	0	$E_{yy}\,ny$	0	$E_{yy}\,ny$	$E_{yy}\,ny$
4,3	0	$E_{yz}\,ny$	0	$E_{yz}\,ny$	0	$E_{yz}\,ny$	$E_{yz}\,ny$
5,1	0	0	$E_{xz}\,nz$	0	$E_{xz}\,nz$	$E_{xz}\,nz$	$E_{xz}\,nz$
5,2	0	0	$E_{yz}\,nz$	0	$E_{yz}\,nz$	$E_{yz}\,nz$	$E_{yz}\,nz$
5,3	0	0	$E_{zz}\,nz$	0	$E_{zz}\,nz$	$E_{zz}\,nz$	$E_{zz}\,nz$
6,1	$E_{xx}\,nx$	0	$E_{xz}\,nz$	$E_{xx}\,nx$	$E_{xx}\,nx$ $+E_{xz}\,nz$	$E_{xz}\,nz$	$E_{xx}\,nx$ $+E_{xz}\,nz$
6,2	$E_{xy}\,nx$	0	$E_{yz}\,nz$	$E_{xy}\,nx$	$E_{xy}\,nx$ $+E_{yz}\,nz$	$E_{yz}\,nz$	$E_{xy}\,nx$ $+E_{yz}\,nz$
6,3	$E_{xz}\,nx$	0	$E_{zz}\,nz$	$E_{xz}\,nx$	$E_{xz}\,nx$ $+E_{zz}\,nz$	$E_{zz}\,nz$	$E_{xz}\,nx$ $+E_{zz}\,nz$
7,1	$E_{xx}\,nx$	$E_{xy}\,ny$	$E_{xz}\,nz$	$E_{xx}\,nx$ $+E_{xy}\,ny$	$E_{xx}\,nx$ $+E_{xz}\,nz$	$E_{xy}\,ny$ $+E_{xz}\,nz$	$E_{xx}\,nx$ $+E_{xy}\,ny$ $+E_{xz}\,nz$
7,2	$E_{xy}\,nx$	$E_{yy}\,ny$	$E_{yz}\,nz$	$E_{xy}\,nx$ $+E_{yy}\,ny$	$E_{xy}\,nx$ $+E_{yz}\,nz$	$E_{yy}\,ny$ $+E_{yz}\,nz$	$E_{xy}\,nx$ $+E_{yy}\,ny$ $+E_{yz}\,nz$
7,3	$E_{xz}\,nx$	$E_{yz}\,ny$	$E_{zz}\,nz$	$E_{xz}\,nx$ $+E_{yz}\,ny$	$E_{xz}\,nx$ $+E_{zz}\,nz$	$E_{yz}\,ny$ $+E_{zz}\,nz$	$E_{xz}\,nx$ $+E_{yz}\,ny$ $+E_{zz}\,nz$
8,1	0	$E_{xy}\,ny$	$E_{xz}\,nz$	$E_{xy}\,ny$	$E_{xz}\,nz$	$E_{xy}\,ny$ $+E_{xz}\,nz$	$E_{xy}\,ny$ $+E_{xz}\,nz$
8,2	0	$E_{yy}\,ny$	$E_{yz}\,nz$	$E_{yy}\,ny$	$E_{yz}\,nz$	$E_{yy}\,ny$ $+E_{yz}\,nz$	$E_{yy}\,ny$ $+E_{yz}\,nz$
8,3	0	$E_{yz}\,ny$	$E_{zz}\,nz$	$E_{yz}\,ny$	$E_{zz}\,nz$	$E_{yz}\,ny$ $+E_{zz}\,nz$	$E_{yz}\,ny$ $+E_{zz}\,nz$

Table 7: δ_{rp} vectors for the 8 pixel corners

usual strain defined by the elastic displacements. The elastic energy then becomes

$$En = \frac{1}{2}\int d^3r\,(\epsilon_\alpha - e_\alpha)\,C_{\alpha\beta}\,(\epsilon_\beta - e_\beta) \tag{21}$$

Substituting for the strain using the linear interpolation scheme described earlier, we can then perform the integration over a single pixel, keeping in mind that the thermal strains are constants over the pixel and are not linearly interpolated. The resulting equation in the nodal displacements is

$$En = \frac{1}{2}u_{rp}D_{rp,sq}u_{sq} + \frac{1}{2}e_\alpha C_{\alpha\beta}e_\beta + T_{rp}u_{rp} \tag{22}$$

where the first term is identical to the case without thermal strains, the second term is a constant quadratic in the thermal strains, and the third term is linear in nodal displacements, with T_{rp} given by

$$T_{rp} = -e_\alpha C_{\alpha\beta}\int d^3r S_{\beta,rp}(x,y,z) \tag{23}$$

So even without periodic boundary conditions, there are terms linear and constant in the displacements. Periodic boundary conditions are very similar to the elastic case studied previously, except for an extra term picked up via the T_{rp} term. In a pixel at the face, edge, or corner of the image, the energy becomes

$$En = \frac{1}{2}u_{rp}D_{rp,sq}u_{sq} + b_{rp}u_{rp} + \frac{1}{2}\delta_{rp}D_{rp,sq}\delta_{sq} + \frac{1}{2}e_\alpha C_{\alpha\beta}e_\beta + T_{rp}u_{rp} + T_{rp}\delta_{rp} \tag{24}$$

where the components of δ_{rp} were given in Table 7, and the u's are the real displacements, brought over from the opposing side or edge or corner of the image in the same way as in the electrical or the no-eigenstrain elastic problems. It is important to note that b_{rp} and δ_{rp} are linear in the applied strains, and T_{rp} is independent of the applied strains.

In the programs THERMAL3D.F or THERMAL2D.F, the whole system size is allowed to change in order find the overall thermal expansion that minimizes the energy of the system. In this case, the applied strains, E_α, are now called the macrostrains, and become dynamic variables that define the size and shape of the periodic unit cell. Their values are determined in the conjugate gradient relaxation process, on an equal footing with the elastic displacements. In the programs, the length of the u vector changes from $ns = nx \times ny \times nz$ to $nss = ns+2$, where $u(ns+1,1)=E_{xx}$, $u(ns+1,2)=E_{yy}$, $u(ns+1,3)=E_{zz}$, $u(ns+2,1)=E_{xz}$, $u(ns+2,2)=E_{yz}$, and $u(ns+2,3)=E_{xy}$.

In subroutine DEMBX, the matrix of second derivatives, or the Hessian matrix, is used to update the gradient and conjugate gradient direction in the relaxation process. For the regular nodal displacements, the stiffness matrices make up the Hessian matrix. However, when the six macrostrains are considered to be variables as well, the Hessian matrix goes from being $(ns,3) \times (ns,3)$ in size to $(ns+2,3) \times (ns+2,3)$ in size. The extra partial derivatives that have to be evaluated are (1) the mixed second derivatives, or the second derivative of the energy with respect to a nodal displacement and a macrostrain, and (2) the second derivative with respect to a macrostrain squared or two different macrostrains.

Figure 2 illustrates schematically what the full Hessian matrix looks like. The upper left-hand corner is the main section, where the second derivatives of the energy are taken with respect to the nodal displacements. Multiplication of a vector by this piece is taken care of in the large DO loop involving the stiffness matrices in subroutine DEMBX. The upper right and lower left-hand parts of the Hessian involve mixed second derivatives with respect to a nodal displacement and a macrostrain. Examining eq. (24), which is the full elastic energy with periodic boundary conditions and thermal energy terms, one can see that the only term that has dependence on both the nodal displacements and the macrostrains is the term $b_{rp}u_{rp}$. The kind of second derivative that is needed is

$$\frac{\partial En}{\partial u_{rp}\partial E_\alpha} \tag{25}$$

In the term mentioned, the only u_{rp} dependence is the explicit one. The parameter b_{rp} depends linearly on E_α, as can be seen in the elastic equivalent of eq. (11). In a linear dependence, the partial derivative with respect to a certain variable is the same as the function evaluated when that variable is one and all other variables are zero. Therefore,

$$\frac{\partial En}{\partial u_{rp}\partial E_\alpha} = b_{rp}(E_\alpha = 1) \tag{26}$$

where all the other macrostrains have been set to zero. This is the technique used in the subroutine BGRAD, which computes the terms needed for these parts of the Hessian matrix.

The bottom right-hand part of the Hessian matrix involves second derivatives with respect to the macrostrains. Again looking at eq. (24), the only term that has quadratic dependence on the macrostrains is what was called the constant term before, that comes from the periodic boundary conditions. In all the finite element programs with fixed macrostrains, this term is designated C. In the two thermal strain programs, this term is expressed by a matrix called *zcon*. The second derivative terms are given simply by the various components of this matrix. The subroutine CONST computes the elements of this matrix, which are constant with respect to nodal displacements and macrostrains.

In subroutine DEMBX, the main part of the multiplication by the Hessian matrix is carried out, and then the other three parts of the matrix multiplication are filled in using subroutine BGRAD and the matrix *zcon*.

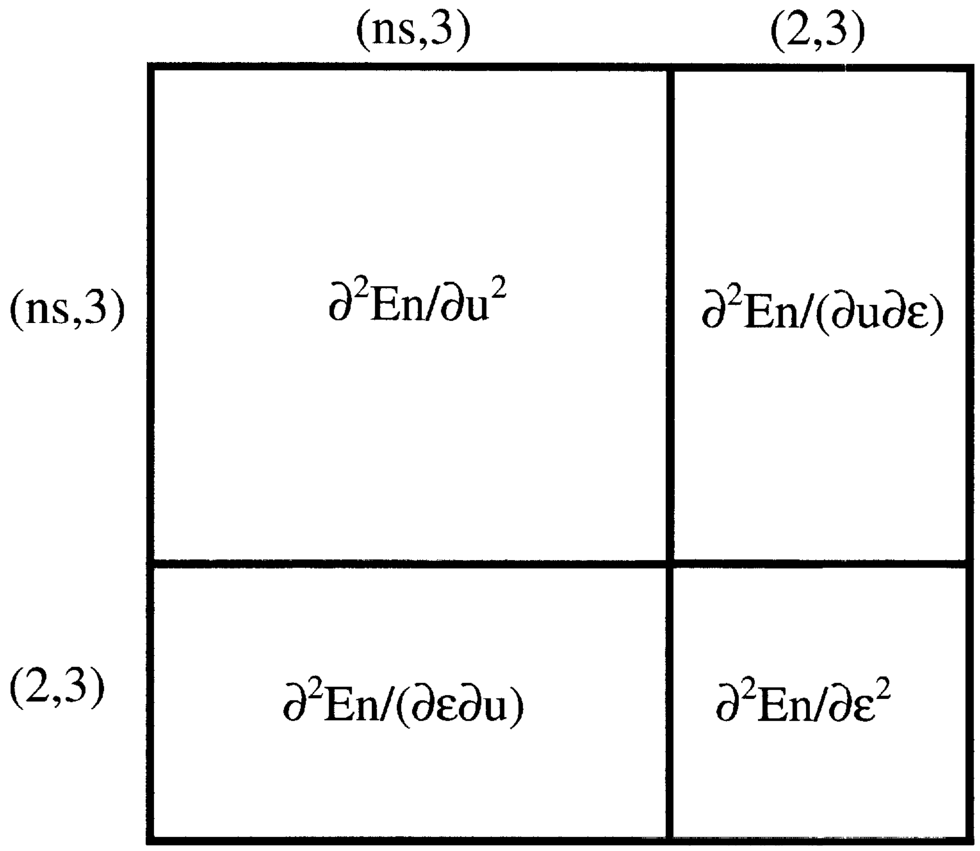


Figure 2: Schematic view of the structure of the Hessian matrix when thermal strains are used (u stands for elastic displacements, ϵ represents the macrostrains).

3 Finite difference theory

3.1 General description, comparison to finite element method

Included in this package of programs are finite difference programs for solving linear conductivity problems on general digital images in 2-D and 3-D, for D.C. and A.C. problems. There are some advantages and disadvantages when comparing finite difference and finite element computations. Regarding the programs described in this manual, the finite difference programs take less memory, as they do not need the integer variable *ib*, which stores neighbor information and is especially needed at the boundaries. This is because the finite difference programs handle the periodic boundary conditions in a different way, as will be discussed below. The finite difference programs in this manual could be written in an energy form, and make use of the second derivative matrix and the variational principle approach in the same way as the finite element formulation. They are not so written in this manual, however.

The way the finite difference algorithm handles boundary conditions between phases is also different from the finite element algorithm. Electrical problems can be handled quite readily with finite differences, but there are some difficulties with elastic problems. At boundaries at which a displacement or stress component takes on a certain value, it is straightforward to write finite difference equations for elastic problems that handle this kind of boundary, using non-centered difference equations. For example, a finite difference elastic code can be written for a porous material, where there is a single kind of solid phase, and so all solid-pore boundaries are zero stress boundaries [8]. Also, the case when the second phase is infinitely stiff, so that the displacement must be the same thoughout connected parts of the phase, can also be handled by finite differences. For a boundary where the displacement and normal force are continuous, however, it is difficult to see how to write a finite difference program that accurately takes both these conditions properly into account. However, two-phase elastic boundaries where both phases have finite moduli can be easily be handled by finite elements.

As for material properties, finite difference electrical conductivity programs can handle arbitrary diagonal conductivity tensors. It is possible to extend them to tensors having non-zero off-diagonal elements, although it makes phase-phase boundaries somewhat more trickier to handle, and requires 2nd and 3rd neighbor information, instead of just nearest neighbor information. Therefore the finite difference programs described in this manual have only diagonal conductivity tensors.

A major advantage of the finite difference approach that we have found is in percolation cases, where a conducting phase becomes sparsely connected. We have found that in simulating continuum percolation cases with digital images, assessing connectivity only through the nearest neighbors agrees rather well with the equivalent continuum calculations [9]. When using finite differences, the only current flow possible is through nearest neighbor connections. In finite elements, however, a node is connected electrically with its first, second, and third nearest neighbors. Therefore a phase could become disconnected in terms of first neighbors, and so disconnected acording to a percolation algorithm, and yet stay connected electrically, if a finite element scheme was being used. So in this case, the finite difference scheme is more physically realistic. If the resolution were infinite, then the result obtained would be the same for the finite difference and finite element methods. Since in a typical problem, and especially so for a sparsely connected problem, the resolution is not infinite, one needs to choose the method that will give more accurate results.

The finite difference programs in this package are: AC3D.F and AC2D.F, for complex (finite frequency or a.c.) problems in 3-D and 2-D, respectively, and DC3D.F and DC2D.F, for d.c. problems in 3-D and 2-D. They are all based on a general digital image, where each phase has a different (and frequency-dependent in the a.c. cases) conductivity. They are not based on a true energy formulation, but rather on a solution of Kirchkoff's laws for the resulting conductance network, which is equivalent to solving Laplace's equation, as is shown next.

3.2 Derivation of finite difference form of Laplace's equation

For a steady state conductivity problem, where the currents are steady in time, the charge conservation equation,

$$\nabla \cdot \vec{j} + \frac{\partial \rho}{\partial t} = 0 \tag{27}$$

becomes simply

$$\nabla \cdot \vec{j} = \nabla \cdot \left(\sigma \vec{E}\right) = 0 \tag{28}$$

or, in regions of constant conductivity,

$$\sigma \nabla^2 V = 0 \tag{29}$$

where V is the position dependent potential or voltage. Between phases having different conductivities, the boundary conditions are that the current normal to the interface and the potential are continuous.

Consider a region of uniform conductivity, and a point (i,j,k) in this region, in the middle of a pixel. If the voltage at (i,j,k) is $u(i,j,k)$, then, to second order in the pixel dimensions, the potential at $i \pm 1$, $j \pm 1$, or $k \pm 1$ is just

$$u(i \pm 1, j, k) = u(i,j,k) \pm a \left[\frac{\partial u}{\partial x}\right]_{i,j,k} + \frac{1}{2} a^2 \left[\frac{\partial^2 u}{\partial^2 x}\right]_{i,j,k} \tag{30}$$

$$u(i, j \pm 1, k) = u(i,j,k) \pm b \left[\frac{\partial u}{\partial y}\right]_{i,j,k} + \frac{1}{2} b^2 \left[\frac{\partial^2 u}{\partial^2 y}\right]_{i,j,k} \tag{31}$$

$$u(i, j, k \pm 1) = u(i,j,k) \pm c \left[\frac{\partial u}{\partial z}\right]_{i,j,k} + \frac{1}{2} c^2 \left[\frac{\partial^2 u}{\partial^2 z}\right]_{i,j,k} \tag{32}$$

where $\Delta x = \pm a$, $\Delta y = \pm b$, $\Delta z = \pm c$, and a, b, and c are the dimensions of the pixel, in some units. Then the finite difference form of Laplace's equation, in terms of the 1-d label m and the six nearest neighbors, can be obtained by adding together the above six equations in pairs and solving for the 2nd derivative terms. The first derivatives cancel out, the gradient squared is formed from the sum of the second derivatives, and Laplace's equation in finite difference form, at each node m, becomes

$$\sum_n \Sigma_{m,n} \left[u(n) - u(m)\right] = 0 \tag{33}$$

where $\Sigma_{m,n}$ is the conductance of the bond connecting node m to its nearest neighbor with nearest neighbor label n, the sum is over the $n = 1, 6$ nearest neighbors, and for convenience, both sides of eq. (33) have been multiplied by the volume of the pixel, abc. The values of Σ_{mn} are $\Sigma_{m1} = \Sigma_{m4} = \sigma\frac{bc}{a}$, $\Sigma_{m2} = \Sigma_{m5} = \sigma\frac{ac}{b}$, and $\Sigma_{m3} = \Sigma_{m6} = \sigma\frac{ab}{c}$, with neighbors (1,2,3) in the (x,y,z) directions, and neighbors (4,5,6) in the ($-x$,$-y$,$-z$) directions. As has been pointed out previously [10], this equation is formally identical to solving Kirchoff's laws of zero net current into each node for this conductance lattice, where the conductances are those of a pixel of material with a given conductivity tensor in a given direction. Therefore formally the length of the pixel appears in the conductance. Note that the finite difference programs are set up for cubic pixels, with $a = b = c = 1$. They can be easily modified for a rectangular parallelipiped pixel, with three arbitrary dimensions. When setting up the conductances, these lengths matter, as the conductances of a given conductivity pixel are different in directions that have a different dimension.

There is an equation like eq. (33) for each node m in the system. Putting all these equations together, a global equation can be written: $A_{mn}\, u_n = 0$. If we build a quadratic form out of this matrix A, $\frac{1}{2}uAu$, then this form is extremized when its gradient with respect to u is zero: $A_{mn}\, u_n = 0$, which is the same as the set of equations we are trying to solve. The formal problem being solved is then the minimization of the quadratic form $\frac{1}{2}uAu$ where the vector u is the voltage vector, and A is a sparse matrix composed of the conductances of all the bonds in the problem. In the finite difference programs, the matrix A is implicitly stored in the vectors gx, gy, and gz. The storage requirements for the finite difference electric programs is much less than those for the finite element programs, and so in this case, the global matrix A is stored. The result of A multiplied times an arbitrary vector can be generated using the vectors gx, gy, and gz, which store the conductances of the problem in the x, y, and z directions respectively. The subroutine PROD does this matrix multiplication, while maintaining periodic boundary conditions and the applied field (see Section 3.3). If there were not the constraint of an applied field, then of course the minimization of a quadratic form such as $\frac{1}{2}uAu$ would be trivially given by $u = 0$.

3.3 Boundary conditions

When two cubic pixels share a face, and are of the same material, the conductance of the bond connecting them is just the conductance of the pixel,

$$\Sigma_m = \sigma_m \cdot (\textit{pixel length}) \tag{34}$$

concentrated into a conducting bond (for a cubic pixel). The situation when two pixels share a face, and are of two different materials A and B, is shown in Fig. 3. Node and pixel j is on the left of the interface, and node and pixel $j + 1$ is on the right side of the interface. In the figure, the conductivity of material A is shown as being different than that of material B. If one were to think of a bond connecting j and $j + 1$, the conductance of that bond is just a series combination of one half the conductance of each pixel,

$$\Sigma = \left(\frac{1}{2\sigma_A} + \frac{1}{2\sigma_B}\right)^{-1} (\textit{pixel length}) \tag{35}$$

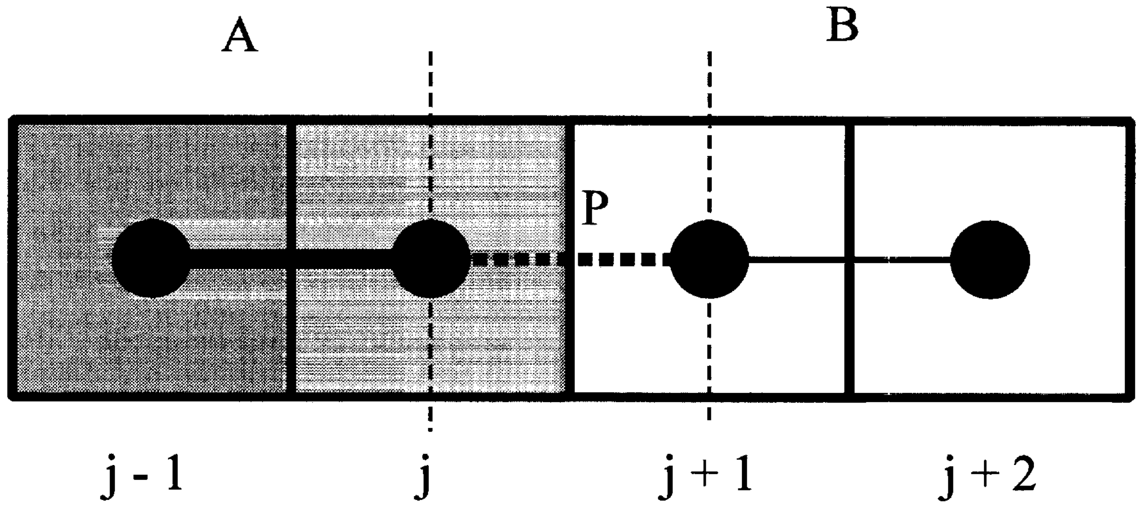


Figure 3: Illustration of nodes near a boundary in the finite difference method.

This construction approximates boundaries. For a pure series situation, this construction is perfectly accurate, and enables the program to give the correct answer. For a curved boundary, like across a spherical inclusion, the answer will not be perfectly accurate. For two adjoining pixels, this construction is, however, the best possible approximation.

The derivation of this condition is simple. Choose the pixel length to be unity, as is done in all the programs. Consider the point P that is exactly on the boundary and halfway between nodes j and $j+1$. The voltage at P is V_P. The boundary condition at the interface is that the normal current across the interface must be continuous. In finite difference terms, that means that

$$\Sigma_A (V_P - V_j) = \Sigma_B (V_{j+1} - V_P) \tag{36}$$

which gives an equation for V_P in terms of V_j and V_{j+1} . Now one must build up the appropriate finite difference equations, but being careful of the boundary. One uses non-centered difference equations for j and $j-1$. The two expansions hold:

$$V_P = V_j + \frac{1}{2}\left[\frac{\partial V}{\partial x}\right]_j + \frac{1}{8}\left[\frac{\partial^2 V}{\partial x^2}\right]_j + \ldots \tag{37}$$

$$V_{j-1} = V_j - \left[\frac{\partial V}{\partial x}\right]_j + \frac{1}{2}\left[\frac{\partial^2 V}{\partial x^2}\right]_j + \ldots \tag{38}$$

Eliminating the first derivative from this pair of equations, and using the boundary condition equation to eliminate V_P, results in the same kind of equation as eq. (33) for node j, but with eq. (35) for the conductivity across the interface. Since in a digital image, all boundaries are locally oriented in the x, y, or z directions, and are essentially 1-D, the formalism in eqs. (35)-(38) above holds generally and is incorporated into the finite difference programs in subroutine BOND.

In all the finite difference programs, periodic boundary conditions are used with the application of an electric field, with its appropriate voltage gradient. Periodic boundary conditions are maintained using a shell of imaginary sites around the main system, as shown in Fig. 4. In Fig. 4, there are nine real sites in this 3×3 pixel system, labelled $1-9$, and 12 imaginary sites, labelled in italics according to which real site they correspond. Suppose that a field is applied from left to right in Fig. 4. The strength of the field is $\frac{1}{3}$, so that there is a voltage step of one placed across the sample. Therefore, the node with italic label 6 has a voltage equal to the voltage at real node 6, but exactly one unit higher. The node with italic label 4 has the same voltage as that at real node 4, but one unit lower. Vertically, since there is no applied field, the nodes with italic labels 2 and 8 have the same voltages as real nodes 2 and 8, respectively. Any update vector coming from the conjugate gradient relaxation routine that is added to the voltage must obey pure periodic boundary conditions (no applied field), so that the inital applied voltage gradient (applied field) is maintained. In subroutine PROD in the finite difference programs, there are DO loops that explicitly maintain this periodicity after every multiplication of the main matrix.

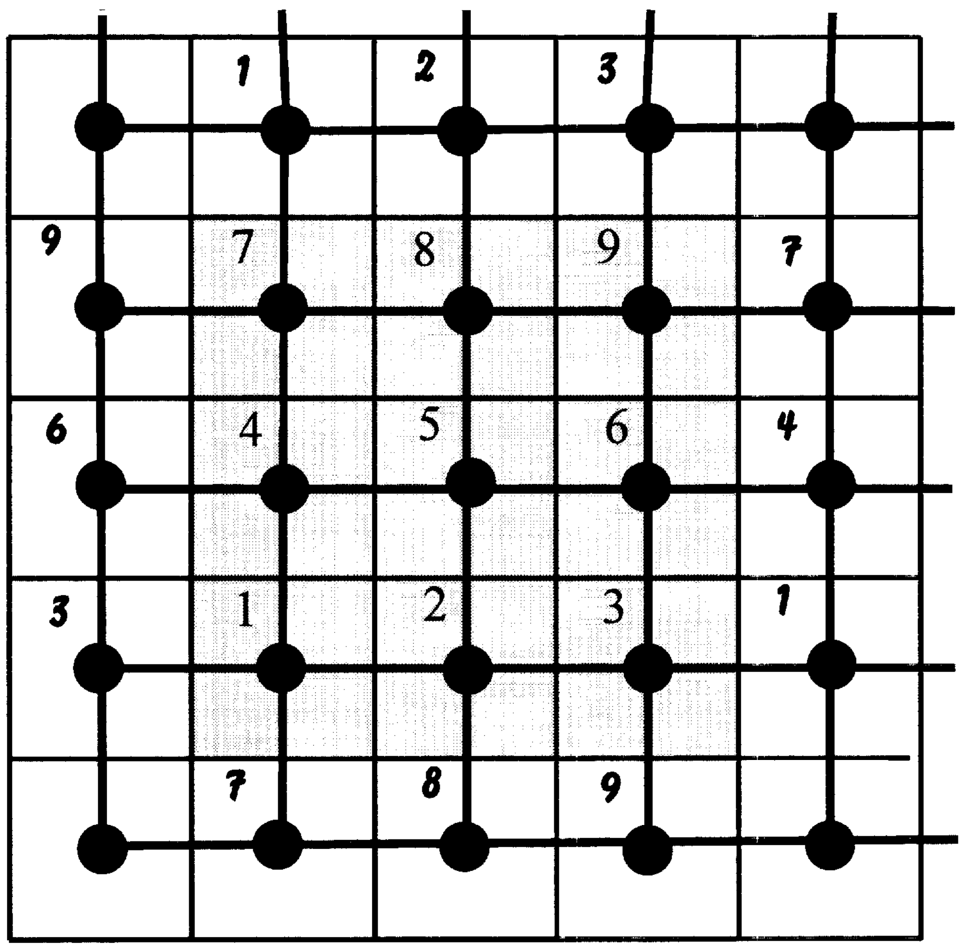


Figure 4: Illustration of how periodic boundary conditions are implemented in the finite difference programs.

4 Explanation of key subroutines and program usage

Key subroutines found in the programs are first explained, before describing how to use the programs themselves. The user who prefers to treat the various subroutines as "black boxes" can skip to Sec. 4.5, which describes the details of actually using the programs.

4.1 Subroutine ENERGY

The subroutine ENERGY is found in all of the finite element programs. It computes the total energy of the system, which is a quadratic functional of the nodal voltages, in the electrical case, and the nodal displacements, in the elastic case. It also computes the gradient of the energy, to be fed into the subroutine DEMBX, which uses the gradient as a starting point for the conjugate gradient routine. The heart of the subroutine is a matrix multiply of the global stiffness matrix times the vector of displacements or voltages. This is accomplished using only the small, local stiffness matrices dk. Essentially, each matrix multiply goes through the same steps that would be necessary for actually building the global stiffness matrix, but does not store the results, which saves tremendously on storage.

4.2 Subroutine DEMBX

Subroutine DEMBX is the conjugage gradient routine in both the finite element and finite difference programs. In both cases, DEMBX is essentially the same, with some details different between the two kinds of programs. In the finite element elastic programs, the only difference from the electric case is the number of components of the vector quantities like displacement. The quadratic energy functional minimized in the finite element programs has explicit linear and constant terms, which come from the periodic boundary conditions. The energy functional in the finite difference programs has only an explicit quadratic term, but the linear and constant terms are implicit in the periodic boundary conditions. A standard conjugate gradient routine is used, which makes use of the Hessian matrix (matrix of partial second derivatives of the energy function with respect to nodal values). The form of the conjugate gradient routine can be found in Section 10.6 of Ref. [5]. Variables in the programs have the same names as in Ref. [5].

4.3 Subroutine CURRENT/STRESS

This subroutine, called CURRENT for the electrical case, and STRESS for the elastic case, computes, respectively, the total current tensor or the total stress tensor in the system. In the finite element case, the averaging is done over each pixel (finite element), using the finite element average current and average stress matrices, and then added up over all the pixels. In the finite difference programs, the total current is added up over all the bonds connecting nodes (see Section 7).

4.4 3-D : 2-D relations for programs

In terms of computer code, the 2-D programs, both finite element and finite difference, were created directly from the 3-D programs. This was done essentially by leaving out the third (z) dimension everywhere. For the finite difference programs, only two conductance vectors, gx and gy, are needed to store the conductance information. For the finite element programs, the stiffness matrices dk are only dimensioned for four nodes, not eight nodes, and the elastic programs have only two degrees of freedom per node, not three as in 3-D. The elastic modulus tensor is 6×6 in 3-D, but only 3×3 in 2-D. For the thermal strain programs, in 3-D the last two entries of the displacement vector contain the six macrostrains, three in each entry. In 2-D, this is also the case, but as there are only three macrostrains in all, the second component of the last entry is not used in the program. The rest of the changes between 2-D and 3-D are obvious, and are in the programs.

Analytically relating 2-D to 3-D elasticity can be a problem. Engineers usually always think in 3-D, and to go to 2-D requires them to think in terms of plane strain (no z strains) or plane stress (no z stresses). However, it is possible to set up the equations of elasticity in 2-D, independent of but analogous to 3-D. This is the approach used in the finite element programs. The result looks like a plane strain approach. If one wishes to use the programs as a plane strain or stress limit from three dimensions, then simply substitute the correct moduli for the 2-D moduli in the programs.

The following formulas are used in the beginning of the finite element elastic programs, and are worth repeating here, for isotropic phases. The subscripts indicate the dimensionality of the modulus. K is the shear modulus and G is the bulk modulus. For isotropic elasticity, the full elastic moduli tensor, C_{ij}, can be expressed in terms of two independent constants. These are usually taken to be either K and G, or E and ν, the Young's modulus and Poisson's ratio, respectively. The 2-D and 3-D relations interrelate the two pairs, as only two of them are independent for isotropic elasticity. In 3-D,

$$\begin{aligned} K_3 &= \frac{E_3}{3(1-2\nu_3)} \\ G_3 &= \frac{E_3}{(1+\nu_3)} \end{aligned} \tag{39}$$

In 2-D,

$$\begin{aligned} K_2 &= \frac{E_2}{2(1-\nu_2)} \\ G_2 &= \frac{E_2}{(1+\nu_2)} \end{aligned} \tag{40}$$

The inverse of these relations is also useful. In 3-D,

$$\begin{aligned} \frac{9}{E_3} &= \frac{1}{K_3} + \frac{3}{G_3} \\ \nu_3 &= \frac{(3K_3 - 2G_3)}{2(3K_3 + G_3)} \end{aligned} \tag{41}$$

and in 2-D,

$$\frac{4}{E_2} = \frac{1}{K_2} + \frac{1}{G_2}$$
$$\nu_2 = \frac{(K_2 - G_2)}{(K_2 + G_2)} \tag{42}$$

4.5 Actual program operation

This subsection gives details of how to actually use the various programs. Further details are in the extensive comments in the programs themselves. In all the programs, the string **(USER)** indicates a place where the user might have to modify the program to fit his particular problem. Possible changes include the value of conductivity or elastic moduli in a phase, the number of phases, and the system size. The Gaussan quadrature program is simple and so explained only in its commments. Sec. 7.7 gives a brief description of the two percolation programs, with operational details left to the program listings, as these programs are simple, too.

4.5.1 Electric and elastic finite element programs

The following gives some actual details of use for the finite element programs ELAS2D.F, ELAS3D.F, ELECFEM2D.F, and ELECFEM3D.F. The eigenstrain programs have some differences in them, and will be discussed separately. The 2-D and 3-D programs are very similar to each other, for both the elastic and electric cases, so that the discussion below assumes 3-D.

The first task is to assign the system size, by choosing the values of nx, ny, and nz. These can be different from each other. The total number of nodes is, in the program, $ns = nx \times ny \times nz$. The comments in the finite element programs show clearly which arrays need to be dimensioned by the system size = total number of nodes. These variables can be dimensioned at the same time with a global replacement, since the dimensions must be changed in each subroutine as well as the main program.

The next task is to generate a digital-image structure in a separate program, or in the subroutine PPIXEL. Define i, j, and k as the indices running in the x, y, and z directions in a right-handed Cartesian coordinate system. Once the structure has been generated, for an image that is $nx \times ny \times nz$, the finite element programs assume it is stored in a file that has been generated by the following piece of code:

```
      DO 10 K = 1, NZ
      DO 10 J = 1, NY
      DO 10 I = 1, NZ

      M = NX*NY*(K - 1) + NX*(J - 1) + I

      WRITE(8,1) PIX(M)
1     FORMAT(i1)
```

```
10   CONTINUE
```

and reads the file in the same way in subroutine PPIXEL. In the programs, this file is called MICROSTRUCTURE.DAT (the name of which can be set by the user). The two OPEN statements in the beginning of the programs give the file names for the microstructure input file (unit=9) and the results output file (unit=7). The values in $pix(m)$ are phase labels, starting at the value one, i.e., phase one has label 1, phase two has label 2, and so on, up to the maximum value of *nphase*, which must be set by the user. If the user wishes to generate a simple structure within the program, the place to do it is in subroutine PPIXEL.

The value of *gtest* is also assigned, and is used for determining when the gradient of the elastic energy is small enough. One usually determines this using solutions of exactly known composite problems (see Section 5), or operating directly on the problem of interest and checking how the various calculated quantites, like effective elastic moduli (conductivity), local stresses (currents), etc. change with relaxation. The value of *gtest* should be determined by numerical experimentation, for the specific problem of interest. The only way to be sure that *gtest* is small enough is to print out the values of the quantity desired and see how many conjugate gradient cycles are enough so that this quantity is no longer changing. If *gtest* is picked to be too small, much CPU time can be wasted in relaxing a system beyond the point at which the desired answer no longer changes significantly.

Material properties have to be assigned to each distinct phase. For the elastic programs, in the isotropic case bulk and shear moduli are assigned in the main program. The variable containing the bulk and shear moduli for each phase is *phasemod*. In the anisotropic case, the entire elastic moduli tensors must be inputted in subroutine FEMAT. In the electric programs, the entire conductivity tensor must be supplied in the main program, for isotropic or anisotropic cases. In ELAS3D.F, the values of the applied strain are assigned (exx, eyy, ezz, exz, eyz, and exy) by the user, and in ELECFEM3D.F, the components of the applied electric field are assigned (ex, ey, ez) (similarly in 2-D).

Sometimes more than one microstructure will be considered in the same program, or the same microstructure but with several different sets of properties will be considered. In this case, the parameter *npoints* can be set to the appropriate value of iterations (*micro* = 1, *npoints*). One then gives an initial set of displacements (voltages). In ELAS3D.F (ELECFEM3D.F), the initial displacements (voltages) are obtained by assuming that the applied strain (field) is the same everywhere. It is sometimes advantageous, in the case when several sets of phase properties are used on the same microstructure, for the initial displacements (voltages) of one computation to be the final values of the last computation. This can save CPU time, and is easy to implement, by only going through the displacement (voltage) initialization if *micro* = some fixed value.

The subroutine FEMAT is then run, and sets up: the stiffness matrix *dk* for each kind of pixel, the elastic moduli tensors *cmod*, and the various linear and constant terms needed because of the periodic boundary conditions. Subroutine DEMBX then runs the conjugate gradient algorithm to find the answer to the problem that satisfies the *gtest* criterion. The user must supply the values (or use the default values) of two parameters, *kmax* and *ldemb*, that control how dembx works. The value of *kmax* controls how many times DEMBX can be called in a given computation, and *ldemb* tells how many conjugate gradient iterations DEMBX will perform at each call. These two parameters can be adjusted so as to be able to see how the relaxation is going. Each time DEMBX returns to the main program, intermediate values of the average stress (current), the energy, and the value of $gb \times gb$ are printed out, to serve as a check on how the relaxation is progressing. If the *gtest* criterion has not been met, DEMBX is then re-entered to perform the next *ldemb* conjugate gradient cycles. When relaxation has finally met the *gtest* criterion, subroutine STRESS (CURRENT) is then used again to output the strain and stress (field and current) at every pixel, in order to compute some kind of average or produce a map of local quantities.

4.5.2 Eigenstrain programs

The eigenstrain programs, THERMAL2D.F and THERMAL3D.F, are quite similar in many ways to the elastic programs but have some significant differences. The main difference is that in THERMAL3D.F, each phase can have an eigenstrain tensor, given by the variable *eigen*. These are 6-vectors (Voigt notation), giving the eigenstrains in the x, y, and z and the shear directions. Usually the shear terms are zero, as the processes that produce eigenstrains do not usually produce shear strains. The programs relax the energy and treat the overall size and shape of the system as being dynamic variables. The final size (given macrostrains in the x, y, and z directions) and shape (shear macrostrains) are given by the interaction between the eigenstrains and elastic moduli of each of the phases. The initial displacements of the problem, along with the macrostrains, can be all set to zero, or perhaps an initial value of the macrostrains can be chosen, by guessing the final answer, and the displacements uniformly strained according to this chosen inital value. The conjugate gradient algorithm is fairly robust, and should give the same answer for any reasonable initial values of the variables.

Subroutine DEMBX, with the help of subroutines CONST and BGRAD for THERMAL3D.F and THERMAL2D.F, runs the conjugate gradient algorithm to find the answer to the problem that satisfies the *gtest* criterion. Subroutine STRESS can then be used to output the stresses and strains of the problem. The values of the macrostrains give the overall expansion and shape change of the system. One should note that if all the shear eigenstrains are initially zero, there will in general be a zero shear macrostrain found.

4.5.3 Finite difference programs

The finite difference electrical conductivity programs can be operated quite similarly to the finite element electrical conductivity programs, with some small differences given below.

(1) Since the finite difference programs only handle diagonal conductivity tensors, the local phase conductivity variable, *sigma*, is not a full tensor but only has space for the diagonal elements.

(2) The subroutine DEMBX is not exited until the *gtest* criterion is met. The total number of conjugate gradient cycles possible within DEMBX is given by *ncgsteps*, a parameter that is set by the user. Intermediate results are printed out every *ncheck* cycles, from within DEMBX.

(3) The main variables, like the voltage vector u, are dimensioned $ns2 = (nx+2) \times (ny+2) \times (nz+2)$, rather than $ns = nx \times ny \times nz$ like in the finite element programs. Also, these dimensions occur explicitly only in the main program, so only have to be changed there.

There are other differences, but these are not in sections that must be changed by the user. Sections 2 and 3 of this manual and the comments in the actual programs should be consulted if the user wishes to know more about these differences.

5 Exact solutions for testing programs

There are many exact solutions of composite problems that are quite useful for testing programs like the ones in this package. Some of these are for special microstructures at general volume fractions, some are for general microstructures with special choices of the phase properties, while the others are for dilute microstructures, where a certain shaped inclusion with different properties is introduced into an initially uniform matrix. A general review of many of these dilute limits can be found in [11, 12, 13]. The relations most useful for testing programs are given in this manual. These kinds of programs can also be useful in exploring the dilute limit for inclusions with shapes that cannot be solved analytically [11].

The importance of having such exact solutions available for checking numerical computations should not be underestimated. It is usually easy to prepare numerical methods for uniform regions. Even including a simple boundary, such as exists in series or parallel problems, is not hard to do. But for random material problems, there are usually many boundaries, of general shape. A proper test of a numerical method will include such boundaries. However, random problems usually have no analytical solution, leaving the problem of how can a numerical result be assessed as to its accuracy? A numerical result for a random system can seem perfectly reasonable, and yet be 100% wrong. The problems discussed in this section give exact results for non-trivial boundaries and choices of phase moduli. These exact solutions can then be used to rigorously test if a numerical method will give the correct answer or not. These solutions can also be used to test things like the effect of resolution (number of pixels describing a microstructural feature) or finite size effects, which result when using periodic boundary conditions and a finite system size to simulate an infinite system.

5.1 Definition of effective properties

The main use of the programs described in this manual are to compute the effective properties of a multi-phase random composite. The effective properties of a composite material are defined simply in terms of the averages of various quantities over the system. First of all, for electric cases we have an applied field, maintained via the periodic boundary conditions, and in elastic cases, we have an applied strain, also maintained by the periodic boundary conditions. Theorems from composite theory assure us that the electric field average and the strain average must equal the applied quantities. In other words, if $\langle ... \rangle$ indicates an average over the entire system,

$$\langle f(\vec{r}) \rangle = \frac{1}{V} \int d^3 r f(\vec{r}) \tag{43}$$

then if, in the electrical case, $\vec{E}$ is the applied field, the average of the microscopic field, which varies from pixel to pixel, is

$$\langle e_p(\vec{r}) \rangle = E_p \tag{44}$$

while in the elastic case, a similar statement holds for the strain, where the applied strain is E_α:

$$\langle \epsilon_\alpha(\vec{r}) \rangle = E_\alpha \tag{45}$$

In addition, an average can be performed over a single phase, denoted by

$$\langle f(\vec{r})\rangle_n = \frac{1}{V_n}\int_n d^3r f(\vec{r}) = \frac{1}{c_n}\langle f(\vec{r})\Theta(\vec{r})\rangle \tag{46}$$

where V_n is the volume of phase n, and c_n is the volume fraction of phase n. The first integral is taken only over the volume of phase n, while the second integral is over the total volume, with $\Theta(\vec{r}) = 1$ for phase n, and zero elsewhere.

There are two equivalent ways to define the effective quantities: an energy method or an average current/stress method. In the energy method, the total energy per unit volume is equated to the energy per unit volume of a uniform medium. The equation used is:

$$En \equiv \frac{1}{2}E_\alpha C_{\alpha\beta}^{eff} E_\beta \tag{47}$$

in the elastic case, and

$$En = \frac{1}{2}E_p\sigma_{pq}^{eff}E_q \tag{48}$$

in the electric case. Various applied fields and strains can be used to pick off various components of the effective quantities.

Another way to define the effective moduli or conductivity is through a stress average or a current density average. For the electric case, the average current density is given by

$$\langle j_p\rangle = \langle\sigma_{pq}e_q\rangle \equiv \sigma_{pq}^{eff}E_q \tag{49}$$

which then defines the effective conductivity, as the average field is just the applied field. For elastic cases, the equivalent expression for the average stress is

$$\langle\sigma_\alpha\rangle = \langle C_{\alpha\beta}\,\epsilon_\beta\rangle \equiv C_{\alpha\beta}^{eff}E_\beta \tag{50}$$

which then defines the effective moduli tensor. The average stress and current formulations are used in the programs, as one run can determine more of the effective properties than in the energy case, where there is only one number, the average energy, produced. The finite element programs do output the energy, however, so that this method can be used. Good review articles of composite theory include Refs. [14, 15]. For the rest of this manual, the superscript eff is dropped, so that properties like conductivity and elastic tensors without a phase label indicate effective properties.

5.2 Series and parallel

The simplest exact composite relations that are sometimes useful are the well-known series and parallel results for two different phases with volume fractions of x for phase 1 and $1-x$ for phase 2. If σ_1 and σ_2 are the isotropic conductivities of the two phases, then if the phases are arranged in parallel , the effective conductivity will be given by $\sigma = x\sigma_1 + (1-x)\sigma_2$. The field in both phases will be uniform and equal to the applied field. If the two phases are arranged in series, then the effective conductivity will be

$$\sigma = \left[\frac{x}{\sigma_1} + \frac{(1-x)}{\sigma_2}\right]^{-1} \tag{51}$$

In this case, the field in each phase will be uniform, but different. The current, however, must be the same in both phases, as the direction of current flow is normal to the interface, so that $\sigma_1\langle\vec{E}\rangle_1 = \sigma_2\langle\vec{E}\rangle_2$. With $x\langle\vec{E}\rangle_1 + (1-x)\langle\vec{E}\rangle_2 = \vec{E}$, it is simple to work out these fields, which can then be used to check the phase average of the field result for the program. It is important that any numerical scheme be checked on more that just the series and parallel problems, as these do not give realistic phase boundaries.

5.3 Small contrast of properties

When the difference between the properties of a two phase material is small, a power series expansion that can be made for the effective properties in terms of this difference becomes useful. A conductivity result which is true for general two phase isotropic microstructures has been derived by Brown [16] and extended by Torquato [14]. The effective conductivity, to second order in the difference $(\sigma_2 - \sigma_1)$, is given by:

$$\sigma = \sigma_1 + c_2(\sigma_2 - \sigma_1) - \frac{1}{d}c_1c_2\frac{(\sigma_2 - \sigma_1)^2}{\sigma_1} + O(\sigma_2 - \sigma_1)^3 + \ldots \tag{52}$$

where d is the dimensionality and c_i is the volume fraction of phase i. The coefficients for the $O(\sigma_2 - \sigma_1)^3$ and higher order terms involve details of the microstructure, and are given in terms of various correlation functions over the phase geometry [14, 16].

A $22 \times 22 \times 22$ pixel cube (phase 2) centered in a $30 \times 30 \times 30$ unit cell was used to test Brown's result for the case of small contrast between the two phase conductivities. For this case, we have $c_1 = 0.39437$, and $c_2 = 0.60563$. We take $\sigma_1 = 1$ always, and vary σ_2 between 1 and 1.3. Figure 5 shows the result, plotted against σ_2, where the quantity $\sigma_1 + c_2(\sigma_2 - \sigma_1)$ has been subtracted from both the numerical and theoretical results. This has been done to show how the numerical results compare to the theoretical results in the quadratic term. The numerical data follows the theoretical line very closely. As the value of σ_2 gets larger, there should also be contributions from the cubic term, which could account for the small differences between the numerical and theoretical results. For cubic symmetry, the effective conductivity tensor is isotropic, so Brown's results can be used to analyze this system.

The same procedure can be carried out for the elastic moduli [17]. A simple way to derive this result, at least up to second order in the modulus contrasts, is to take Hashin's bounds [15], which bound the effective properties between an upper and lower limit, and expand them to second order in the modulus differences. These bounds are known to be exact to second order in the modulus differences, and in fact the upper and lower bounds agree exactly to this order. The 3-D results for the effective bulk modulus K and shear modulus G are:

$$K = K_1 + c_2(K_2 - K_1) - \frac{c_1c_2}{(K_1 + \frac{4}{3}G_1)}(K_2 - K_1)^2 + O(K_2 - K_1)^3 + \ldots \tag{53}$$

$$G = G_1 + c_2(G_2 - G_1) - \frac{2c_1c_2(K_1 + 2G_1)}{5G_1(K_1 + \frac{4}{3}G_1)}(G_2 - G_1)^2 + O(G_2 - G_1)^3 + \ldots \tag{54}$$

In 2-D, the results are

$$K = K_1 + c_2(K_2 - K_1) - \frac{c_1c_2}{(K_1 + G_1)}(K_2 - K_1)^2 + O(K_2 - K_1)^3 + \ldots \tag{55}$$

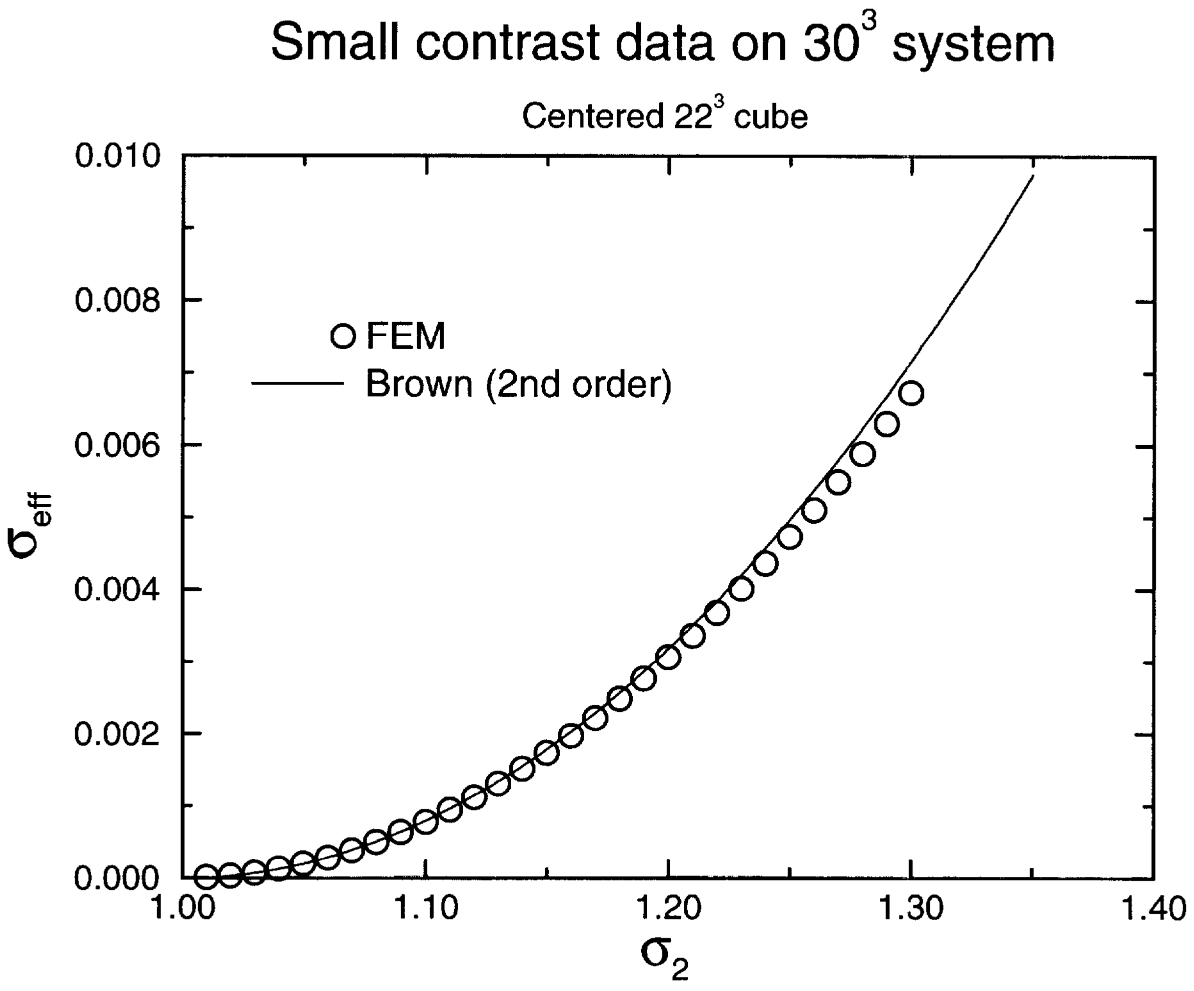


Figure 5: Showing σ vs. σ_2 when σ_2 differs only slightly from σ_1. The points are finite element data, and the straight line is Brown's exact expansion to second order in the contrast $(\sigma_2 - \sigma_1)$.

$$G = G_1 + c_2(G_2 - G_1) - \frac{c_1 c_2 (K_1 + 2G_1)}{2G_1(K_1 + G_1)}(G_2 - G_1)^2 + O(G_2 - G_1)^3 + ... \tag{56}$$

Similar results as those for the small contrast in conductivity case are obtained when testing the accuracy of how well the finite element method computes the small contrast in elastic moduli case. The bounds themselves [15] can also be useful for checking results, since whatever effective elastic moduli are found for an n-phase composite must lie between the appropriate n-phase bounds. Just as in the electric case, the coefficients for the $O()^3$ and higher order terms involve details of the microstructure, and are given in terms of various correlation functions over the phase geometry [17].

5.4 Keller and Mendelsen 2-D result for conductivity

Another general conductivity result, due to Keller and Mendelson [18, 19], for a general 2-D isotropic two phase microstructure, is that

$$\sigma(\sigma_1, \sigma_2)\sigma(\sigma_2, \sigma_1) = \sigma_1\sigma_2 \tag{57}$$

where $\sigma(\sigma_1, \sigma_2)$ means the effective conductivity obtained when phase 1 has conductivity σ_1 and phase 2 has conductivity σ_2 , and $\sigma(\sigma_2, \sigma_1)$ means the phase conductivities have been interchanged before determining the effective conductivity.

5.5 Field fluctuation result

For isotropic two component mixtures, in 2-D or 3-D, there are exact results connecting field fluctuations and the effective conductivity [20]. For elastic problems, one can also exactly relate strain averages to the effective bulk and shear moduli [20], though this topic is not discussed in this manual.

For conductivity, one must first formulate the average over a phase:

$$\langle f \rangle_j = \frac{1}{c_j}\langle \Theta_j f \rangle \tag{58}$$

where Θ_j is equal to 1 inside phase j, and zero elsewhere, and c_j is the volume (area) fraction of phase j, $j = 1, 2$. Plain brackets indicate an average over the whole system.

The exact relation can then be stated simply:

$$\frac{\langle E^2 \rangle_j}{\langle E^2 \rangle} = \frac{1}{c_j}\frac{\partial \sigma}{\partial \sigma_j} \tag{59}$$

where again σ is the effective conductivity. If σ is known analytically as a function of σ_j, then this differentiation can be readily carried out. If not, this derivative can always be evaluated numerically, by evaluating σ for $\sigma_j \pm \delta$, with δ being a small number.

Consider a system where σ is analytically known. The checkerboard microstructure, as shown in Fig. 6, can be evaluated by the Keller-Mendelson formula given above. Since the microstructure is clearly invariant under inversion of σ_1 and σ_2, the effective conductivity must then be $\sigma = (\sigma_1\sigma_2)^{1/2}$.

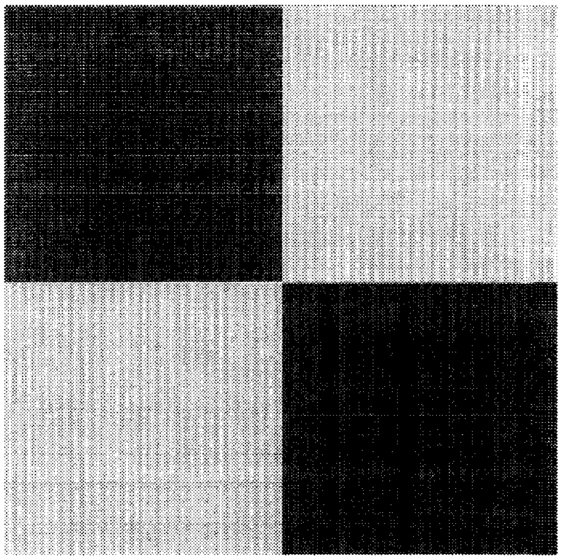


Figure 6: Showing the checkerboard microstructure, with dark gray being phase 2 and light gray phase 1.

The exact formula then gives $\frac{\langle E^2\rangle_2}{\langle E^2\rangle} = (\sigma_1/\sigma_2)^{1/2}$. Fig. 7 shows the numerical results, compared to the exact theoretical results, for a 128×128 system for a variety of conductivity ratios. Excellent agreement is shown, with some systematic disagreement growing at larger values of the conductivity difference.

If we fix the conductivity ratio between the two phases to be 10, we can then examine how well the field averages and effective conductivity are computed as a function of system size $L \times L$. Table 8 gives the data obtained.

5.6 Equal shear modulus

A result that can be very useful in testing elastic programs is the equal shear modulus result, true for any microstructure [21, 22]. If there are only two phases present, with equal shear moduli G but different bulk moduli K_1 and K_2, then the effective shear modulus of the entire system is just G, and the effective bulk modulus K of the system is, in 2-D,

$$K = \frac{G(c_1K_1 + c_2K_2) + K_1K_2}{G + c_1K_2 + c_2K_1} \tag{60}$$

and in 3-D,

$$K = \frac{\frac{4}{3}G(c_1K_1 + c_2K_2) + K_1K_2}{\frac{4}{3}G + c_1K_2 + c_2K_1} \tag{61}$$

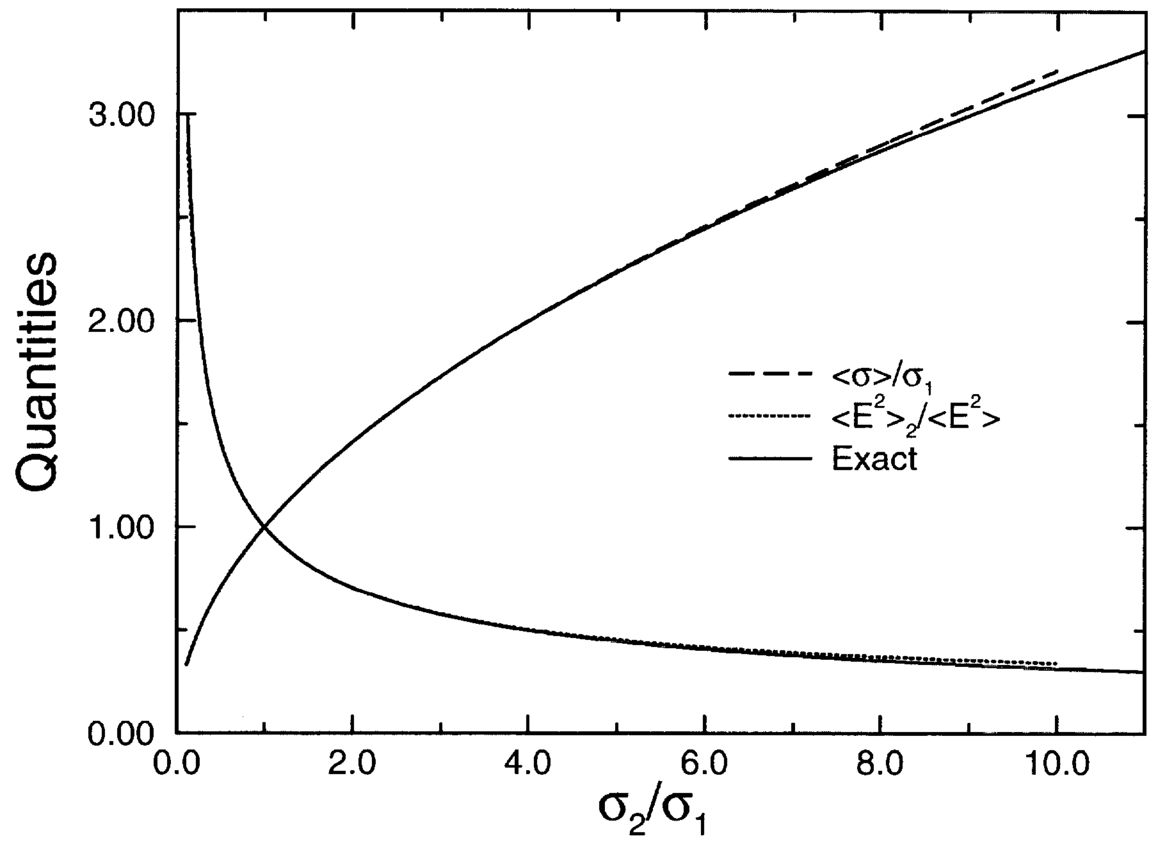


Figure 7: Showing the effective conductivity and average of the electric field magnitude squared in phase 2 for the checkerboard, as a function of σ_2. The points are numerical finite element results, and the lines are the exact results discussed in the text. The system size was 128 × 128.

L	σ/σ_1(FEM)	σ/σ_1(Exact)	% Error	$\frac{\langle E^2\rangle_2}{\langle E^2\rangle}$ (FEM)	$\frac{\langle E^2\rangle_2}{\langle E^2\rangle}$ (Exact)	% Error
16	3.4442	3.1623	8.9	0.4132	0.3162	30.7
32	3.3232	3.1623	5.1	0.3791	0.3162	19.9
64	3.2550	3.1623	2.9	0.3569	0.3162	12.9
128	3.2159	3.1623	1.7	0.3424	0.3162	8.3
256	3.1934	3.1623	1.0	0.3330	0.3162	5.3
512	3.1804	3.1623	0.6	0.3269	0.3162	3.4
1024	3.1728	3.1623	0.3	0.3230	0.3162	2.1

Table 8: Size dependence ($L \times L$ system) of effective conductivity and field average for checkerboard, as determined by finite element method. Each individual "check" of the checkerboard is $L/2 \times L/2$. The conductivity ratio $\sigma_2/\sigma_1 = 10$.

where c_1 and c_2 are the volume fractions of the two phases. Note that the only difference between the two equations is the 4/3 factor in front of the shear modulus G. In 2-D, the result is somewhat simpler if expressed in terms of the Young's modulus E and the Poisson's ratio ν, $E = c_1 E_1 + c_2 E_2$, and $\nu = c_1 \nu_1 + c_2 \nu_2$ [22], which actually looks like a parallel result but is valid for general two phase microstructures.

We choose the same cubic image as was used for the small contrast in conductivity case to test the equal shear moduli result. A $22 \times 22 \times 22$ pixel cube (phase 2) is centered in a $30 \times 30 \times 30$ unit cell, so that the volume fractions are: $c_1 = 0.60563$ and $c_2 = 0.39437$. The two shear moduli are taken equal to unity, $G_1 = G_2 = 1$, and the two bulk moduli are $K_1 = 1$, and $K_2 = 20$. The exact answer, according to eq. (53) is $K = 2.263250$, while the numerical answer is $K = 2.269684$, a difference of only 0.3%.

In 2-D, a test of the equal shear moduli result can be combined with a test of the effect of digital resolution by considering the checkerboard problem. The shear moduli are both equal to 2, and there is a ratio of 10 between the bulk moduli ($K_1 = 1$, $K_2 = 10$) (see Fig. 6 for a picture of the microstructure). The effective bulk modulus is computed as a function of system size $L \times L$, where each "check" is $L/2 \times L/2$. The exact value of K is 2.8 using eq. (60). Table 9 displays the data from this test. For $L \geq 16$, the error in the effective bulk modulus is about 1% or less.

System size	K	% diff.
2	5.500	96.4
4	3.271	16.8
8	2.937	4.9
16	2.841	1.5
32	2.812	0.4
64	2.804	0.1
128	2.8012	0.04
256	2.8003	0.01
512	2.8001	0.004

Table 9: Effective bulk modulus for checkerboard–equal shear moduli case.

5.7 Intrinsic properties for spheres, circles, and cubes

A useful dilute limit is that of spherical (circular) inclusions randomly distributed in a matrix. To be in the dilute limit, the volume fraction of the inclusion phase should probably be less than or equal to 5%, although this limit can be higher or lower, depending on inclusion shape and contrast of properties with the matrix. Call the inclusion phase 2. In general we find that, for some property being considered, say F, the property in the dilute limit has the form

$$F = f_1 \left(1 + [F]c_2\right) \tag{62}$$

where [F] is called the intrinsic property [11], and $F = f_1$ when $c_2 = 0$. [F] is a function of the shape of the particle, and the contrast between its properties (f_2) and the properties of the matrix (f_1). This is true for any shape inclusion that has been averaged over orientation. For the conductivity problem, for spherical inclusions in 3-D, the intrinsic conductivity $[\sigma]$ is [11]

$$[\sigma] = \frac{3(\sigma_2 - \sigma_1)}{(2\sigma_1 + \sigma_2)} \tag{63}$$

and the intrinsic elastic moduli are given by [23]

$$[K] = \frac{\left(K_1 + \frac{4}{3}G_1\right)\left(\frac{K_2}{K_1} - 1\right)}{K_2 + \frac{4}{3}G_1} \tag{64}$$

$$[G] = \frac{5\left(K_1 + \frac{4}{3}G_1\right)(G_2 - G_1)}{3G_1\left(K_1 + \frac{8}{9}G_1\right) + 2G_2(K_1 + 2G_1)} \tag{65}$$

For circular inclusions in 2-D, the corresponding equations for the conductivity and for the elastic moduli are [24]

$$[\sigma] = \frac{2(\sigma_2 - \sigma_1)}{(\sigma_1 + \sigma_2)} \tag{66}$$

$$[K] = \frac{(K_1 + G_1)\left(\frac{K_2}{K_1} - 1\right)}{K_2 + G_1} \tag{67}$$

$$[G] = \frac{2(K_1 + G_1)(G_2 - G_1)}{G_1(K_1 + G_1) + G_2(K_1 + G_1)} \tag{68}$$

Note the similar structure between these sets of formulas in 2-D and 3-D.

To test a finite difference or finite element program, one generally puts one particle in the matrix, with a unit cell-size chosen so that the volume fraction is small enough to be in the dilute limit. Even with periodic boundary conditions, the conductivity term or elastic moduli term that is linear in the volume fraction of the inclusion phase is unchanged from the infinite system limit. Refs. [11] and [12] contain electric, elastic, and viscosity dilute limits for many other shape particles.

For the sphere, we use a $40 \times 40 \times 40$ unit cell, and a 15 pixel diameter sphere, centered on the middle of a pixel, so that the sphere volume fraction is 0.028. Figure 8 shows the results for the intrinsic conductivity. The solid line is the exact solution, eq. (63). For x close to zero, it appears that the finite element is more accurate, while for $x >> 1$, the finite difference seems to be more accurate. To test the effect of resolution on the result, the results at $x = 10$, the case with the worst accuracy, were re-run for a 31 pixel diameter sphere in a 100^3 unit cell. This made the volume fraction slightly smaller, which would also tend to slightly improve the results. The numbers in Table 10 in parentheses are these results. For a factor of 2 improvement in resolution, the absolute error in $[\sigma]$ decreased by a factor of three for the finite element method, and by a factor of eight for the finite difference method.

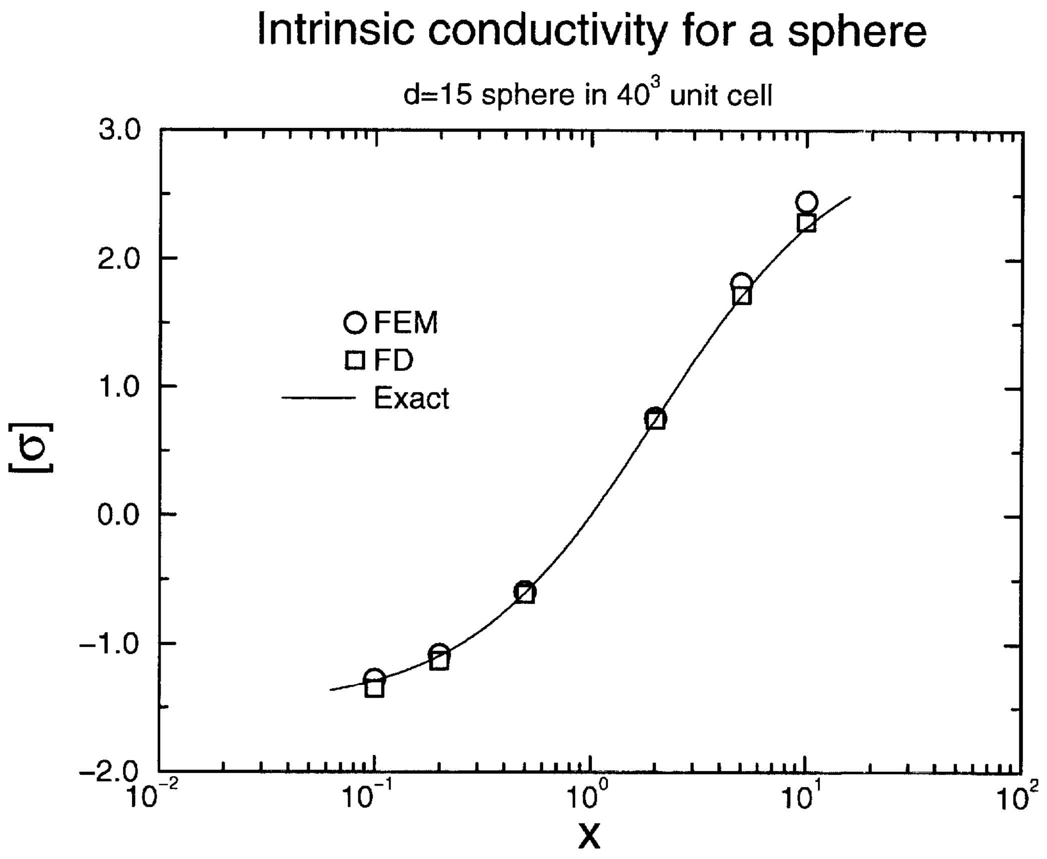


Figure 8: Intrinsic conductivity for a 15 pixel diameter sphere embedded in a 40^3 unit cell, as a function of the ratio of the sphere conducticity to the matrix conductivity. Finite element and finite difference data and the exact result are compared.

x	FEM	| % Error |	FD	| % Error |	Exact
0.1	-1.342	4.4	-1.382	7.5	-1.286
0.2	-1.121	2.8	-1.146	5.0	-1.091
0.5	-0.6029	0.5	-0.6085	1.4	-0.600
2	0.7653	2.0	0.7577	1.0	0.750
5	1.861	8.6	1.812	5.7	1.714
10	2.576 (2.353)	14.5 (4.6)	2.471 (2.278)	9.8 (1.2)	2.25

Table 10: Intrinsic conductivities for sphere, comparing the finite element and finite difference techniques.

The result for the finite difference case only differs by 1.2% from the exact value of 2.25 in this case.

For the same system, we can also calculate the intrinsic bulk and shear moduli. Choosing $K_1 = 1$ and $G_1 = 0.75$, and keeping, for simplicity, $K_2/K_1 = G_2/G_1 = x$ (so that the same Poisson's ratio is maintained in inclusion and matrix), it turns out that using eqs. (64) and (65), both intrinsic moduli are the same and are equal to $2(x-1)/(x+1)$. Figure 9 shows the numerical and exact results for both intrinsic moduli, for a 15 pixel diameter sphere in a 40^3 unit cell. The actual values are given in Table 11. The errors are qualitatively similar to the intrinsic conductivity case. The $x = 10$ point was rerun for the 31 pixel diameter sphere in a 100^3 unit cell, and the result is shown in Table 11 in parentheses. For both the intrinsic bulk and shear moduli, the absolute error decreased by almost exactly a factor of two for the factor of two increase in resolution. There was little difference in accuracy between the computation of $[K]$ and $[G]$.

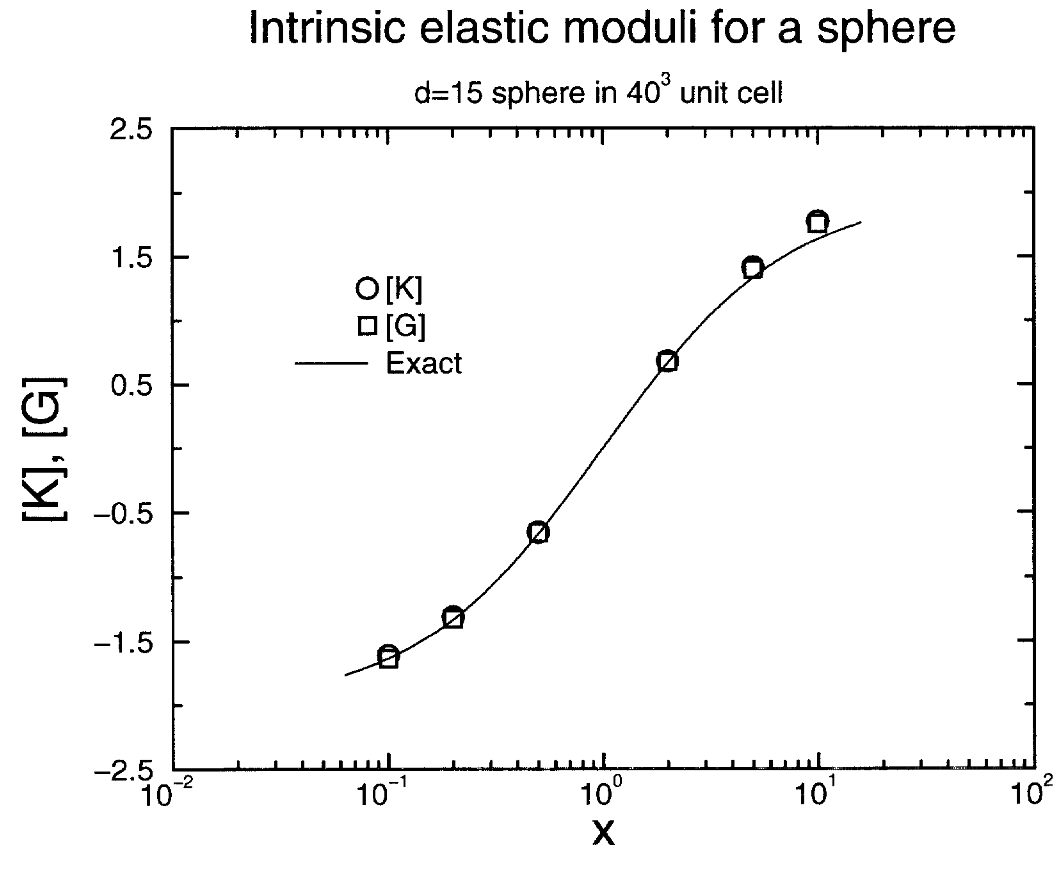


Figure 9: Intrinsic elastic moduli for a 15 pixel diameter sphere embedded in a 40^3 unit cell, as a function of the ratio of the sphere Young's modulus to the matrix Young's modulus. The three sets of data show [K], [G], and the exact result, which is the same for both intrinsic moduli.

In 2-D (to limit computational effort), we can further analyze the resolution dependency of the finite element calculation of the intrinsic elastic moduli. A circle is placed in a

x	[K]	| % Error |	[G]	| % Error |	Exact
0.1	-1.612	1.5	-1.612	1.5	-1.636
0.2	-1.31	1.8	-1.328	0.4	-1.333
0.5	-0.658	1.3	-0.662	0.7	-0.667
2	0.682	2.3	0.678	1.7	0.667
5	1.414	6.0	1.398	4.9	1.333
10	1.774 (1.706)	8.4 (4.3)	1.755 (1.697)	7.3 (3.7)	1.636

Table 11: Intrinsic elastic moduli for sphere, $d = 15$ in 40^3 system

computational cell, with a diameter of 1/10'th the cell size. The intrinsic moduli for the bulk and shear moduli are then computed as a function of cell size. The circle moduli are ten times larger than those of the matrix, with the same Poisson's ratio. Table 12 shows the data for this computation.

System size	[K]	% diff.	[G]	% diff.
20	1.548	10.1	1.318	6.9
40	1.548	10.1	1.362	12.9
80	1.475	4.9	1.290	4.6
160	1.447	2.9	1.261	2.3
320	1.433	1.9	1.251	1.5
640	1.425	1.3	1.244	0.9

Table 12: Intrinsic elastic moduli for circle–effect of digital resolution, using finite element method

Another test of the accuracy of both the finite element and finite difference electrical programs is to run the example of the dilute limit of a cubical shaped inclusion. Since cubic pixels can replicate the shape of a cubical inclusion exactly, then any effect of a digitally rough boundary (imperfect representation of the geometry) can be dispensed with. Eyges [25] has done a very careful numerical treatment of this problem, which can be used as a check on these programs. We use a $10 \times 10 \times 10$ cube, centered in a $40 \times 40 \times 40$ unit cell, so that the cube volume fraction was 0.0156. The conductivity of the matrix, σ_1, is taken to be unity, with the conductivity of the inclusion, σ_2 , ranging between 0.1 and 10. For a cubic symmetry system, the conductivity tensor is isotropic, so there is only one independent diagonal component, and no off-diagonal components. Let $x = \sigma_2/\sigma_1$. Figure 10 shows the results. It is interesting to note that, for high values of x, the finite difference result actually tends to be a bit more accurate than does the finite element result. It is known that in the limit of very high values of x, the finite element program tends to be about 8-9% high for the dilute limit of cubes [11]. The formula used to plot Eyges' [25] data was

$$[\sigma(x)] = \frac{0.486(x-1)^2 + (x-1)}{1 + 0.820(x-1) + 0.143(x-1)^2} \tag{69}$$

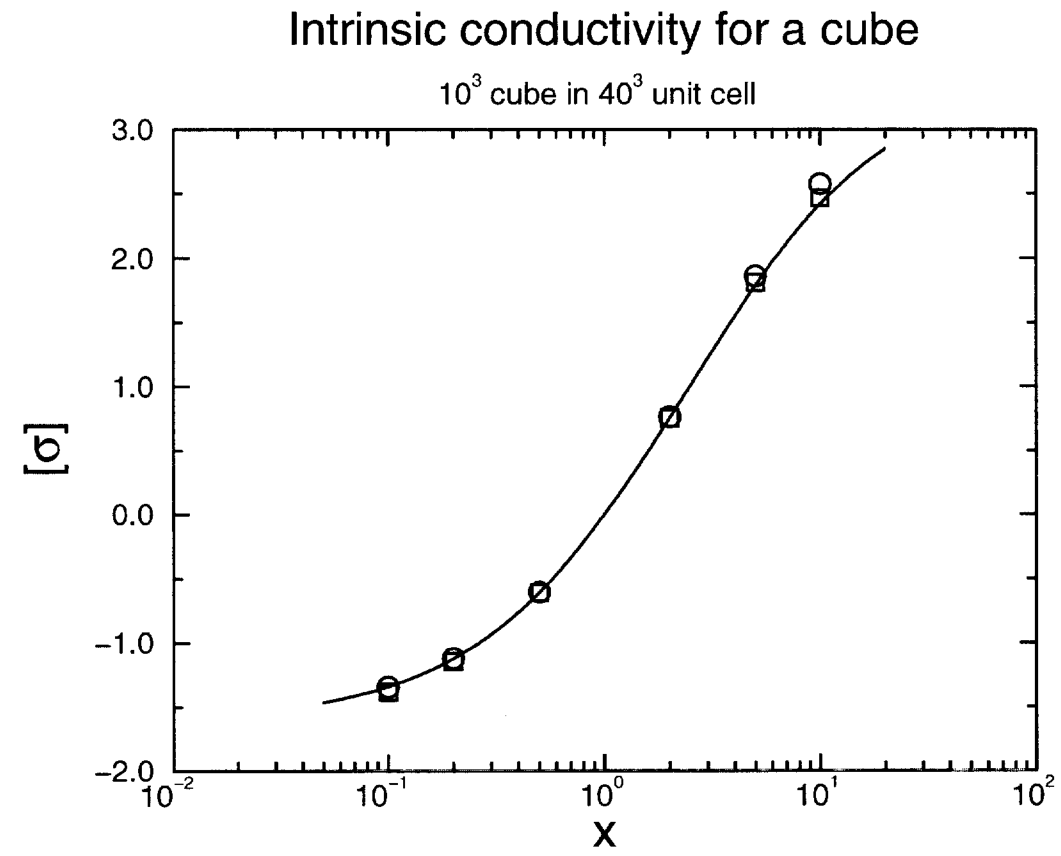


Figure 10: Intrinsic conductivity for a 10^3 cube embedded in a 40^3 unit cell, as a function of the ratio of the cube conductivity to the matrix conductivity. The three sets of data (circle, square, line) compare the finite element method, the finite difference method, and Eyges' [25] data, respectively.

5.8 Vegard's law and Goodier result for thermal strains

When thermal strains (eigenstrains) are present, there are several other sets of exact results available to check the output of the programs THERMAL3D.F and THERMAL2D.F. Suppose first that for all n phases, each phase has the same elastic moduli but different isotropic thermal strains, with the thermal strain in each direction being e_i for the i'th phase. Then the overall system strain in any direction will be simply $\epsilon = \Sigma c_i e_i$, $i = 1, n$. This result is known in the physics community as Vegard's law [26]. This result is true for any microstructure, in both two and three dimensions. Actually, the thermal strain tensor need not be isotropic. The overall thermal strain tensor will be the sum of the volume fraction-weighted phase thermal strain tensors.

A related result, not as well-known, by Goodier [27] applies to this same case where there are only two isotropic phases. In this case, the trace of the strain in the inclusion phase (3-D) is simply

$$Tr\,\epsilon = e\frac{(1+\nu_3)}{(1-\nu_3)} \tag{70}$$

where the thermal strains in any direction are $e_i = e$, the trace does not include thermal strains, and ν_3 is the 3-D Poisson's ratio. The strain trace is constant inside the inclusion. In 2-D (plane strain) the result is

$$Tr\,\epsilon = e(1+\nu_2) = \frac{e}{(1-\nu_3)} \tag{71}$$

where ν_2 is the 2-D Poisson's ratio. Equations (39)-(42) can be manipulated to show the relationship between ν_2 and ν_3.

The trace of the stress in the inclusion is then also constant, and is simply given by (3-D)

$$Tr\,\sigma = 3K_3(Tr\,\epsilon - 3e) = 3K_3 e\left[\frac{(1+\nu_3)}{(1-\nu_3)} - 3\right] \tag{72}$$

and in 2-D,

$$Tr\,\sigma = 2K_2(Tr\,\epsilon - 2e) = 2K_2 e[(1+\nu_2) - 2] \tag{73}$$

where K_3 and K_2 are the 3-D and 2-D (plane strain) bulk moduli, respectively.

It is known that if an elliptical (2-D) or ellipsoidal (3-D) inclusion is introduced into a material, where the inclusion and matrix have the same elastic moduli but different thermal strains, the individual components of the stress tensor are uniform inside the inclusion [7, 27]. Recently, a conjecture has been made that, in 2-D, if the inclusion is in the shape of a equilateral 5-pointed star, the stress tensor components inside the star will also be uniform [28]. The Goodier result [27] of course predicts that at least the trace of the stress tensor will be uniform in any inclusion.

This conjecture about a 5-pointed star can be tested using the program THERMAL2D.F. A digital image can be made of an inclusion in a matrix, where the inclusion has arbitrary shape. With the properties assigned, the system can be relaxed and a stress picture can be made, along with a picture of the trace of the stress tensor. In most case, it should be obvious whether a quantity is uniform within the inclusion or not. In addition, the average stress and its standard deviation can be computed within the inclusion. If a quantity is indeed uniform, then the standard deviation should be much smaller than the average, and reflect only things like finite size effects, and using square pixels to approximate curved and angular boundaries.

The top images in Fig. 11 show an elliptical inclusion, with an aspect ratio of three. The stress map on the left is of $-\sigma_{xx}$, where the stress levels, from high to low, are red-green-white-black (top to bottom in the accompanying color bar). This component of the stress appears to be uniform inside the inclusion, as it is supposed to be. The 2- D moduli

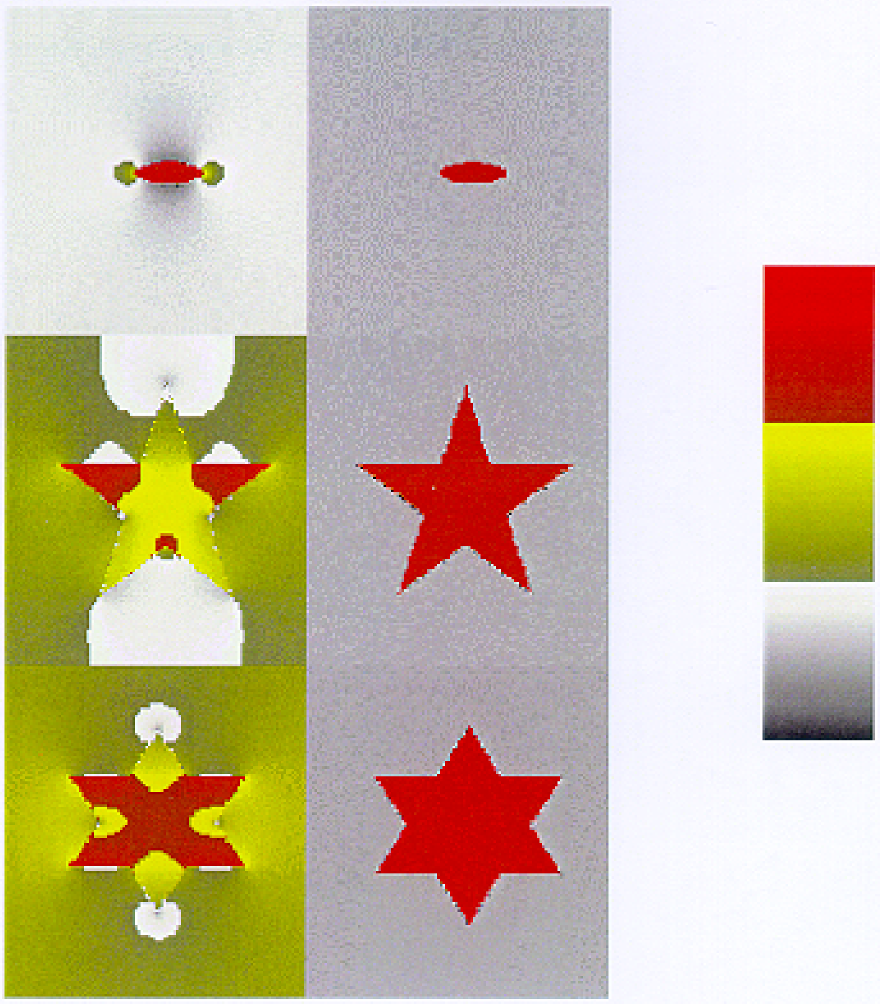


Figure 11: Thermal stress for inclusions with the same elastic moduli as the matrix, but different thermal eigenstrain. Stress is high (top) to low (bottom) in color bar). Left: $-\sigma_{xx}$, right: negative of the trace of the stress tensor. Images from top: Ellipse, 5-pointed star, 6-pointed star.

used were: $E_2 = 1$, $\nu_2 = 0.3$, $K_2 = 5/7$, and $G_2 = 5/13$, in arbitrary units. The inclusion had a thermal strain of $\epsilon_{xx} = \epsilon_{yy} = 1$, and the matrix had zero thermal strain. The right top image shows the negative trace of the stress tensor, which is also clearly uniform inside the inclusion. The stress in the inclusion is negative (compressive), since a positive thermal strain, constrained by the outer matrix, results in a compressive stress inside the inclusion (see Section 2.4).

The left image in the 2nd row from the top of Fig. 11 shows a map of $-\sigma_{xx}$, with the same choice of elastic parameters, for a 5-pointed star-shaped inclusion. This component of the stress tensor is clearly non-uniform in the inclusion. The right hand image shows the negative trace of the stress tensor, which certainly appears uniform, according to Goodier's result. With the choice of elastic parameters used, the trace of the stress tensor is predicted to be, inside the inclusion, equal to -1. For a 512×512 unit cell, and the area fraction of the

BLANK PAGE

star being 7.6%, the trace of the average stress tensor inside the inclusion is −0.93, in error by only 7%. This difference reflects the finite extent of the system, since the Goodier result is for an infinite system, and errors at the star boundary, whose straight lines are approximated by a digital boundary. A similar finite element computation was also carried out recently [29], with similar results. The remaining row in Fig. 11 shows analogous computations for a six-pointed star, with results similar to that of the 5-pointed star.

5.9 Hashin and Rosen thermal strain result

A result by Hashin and Rosen [30], which is true for both 2-D and 3-D, exactly relates, for the case of two phases, the effective bulk modulus K and the effective thermal strain e when the two phases do not have the same elastic moduli. The isotropic thermal strain for phase i is e_i, and the effective isotropic thermal strain for the composite system is denoted e. The result for the value of e is

$$e = c_1 e_1 + c_2 e_2 + \frac{(e_2 - e_1)}{\left(\frac{1}{K_2} - \frac{1}{K_1}\right)} \left[\frac{1}{K} - \frac{c_1}{K_1} - \frac{c_2}{K_2}\right] \tag{74}$$

and is true for isotropic components and an isotropic composite. There are also versions of this equation valid for anisotropic components and composites [30]. It is easy to combine this equation with that for the dilute limit bulk modulus for spherical inclusions to get the dilute limit thermal strain for a spherical inclusion that has different elastic properties than does the matrix. This result, for an inclusion labelled "2" embedded in a matrix labelled "1", is

$$[e] = \frac{\frac{K_2}{K_1}(K_1 + \frac{4}{3}G_1)(e_2 - e_1)}{(K_2 + \frac{4}{3}G_1)} \tag{75}$$

To illustrate the accuracy with which the finite element programs obey the Hashin-Rosen result, eq. (74), a 200 × 200 pixel system was set up. A total of 160 overlapping circles, 15 pixels in diameter, were thrown down randomly, such that the matrix phase had an area fraction of $c_1 = 0.4779$, and the inclusion phase had an area fraction $c_2 = 0.5221$. The matrix phase had $E_1 = 1.0$, $\nu_1 = 0.2$, and a thermal strain of 0.1 in the x and y directions ($e = 0.1$). The inclusion phase had $E_2 = 0.3$, $\nu_2 = 0.1$, and a thermal strain of 0.4 in the x and y directions ($e = 0.4$). Using the Hashin-Rosen equation, an overall expansion of 0.2109 is predicted. The program THERMAL2D.F gave a value of 0.2103 in the x- direction, 0.2112 in the y-direction, and an average expansion of 0.2108. This average value is within 0.05% of the exact value. The small difference between the x and y directions (< 0.5%) is due to the fact that this fairly small numerical system is not exactly isotropic.

5.10 Mackenzie result for pressurized pore space

A useful exact result has been derived by Mackenzie [31], for an isotropic porous solid filled with a gas or liquid at a pressure p. The expansion of the material is given exactly by an equation involving only the bulk modulus of the solid frame, K_s , the effective bulk modulus of the porous solid, K, and the pressure p. This is analogous to the Hashin-Rosen result, where only the effective bulk modulus of the composite had to be known in order to be able to compute the effective thermal expansion. The strain ϵ is exactly given by:

$$\epsilon = \frac{p}{3}\left(\frac{1}{K} - \frac{1}{K_s}\right) \tag{76}$$

In 2-D, simply replace the "3" by "2" in eq. (73), and use the 2-D bulk moduli (see Sec. 6.3 for suggestions on implementing this computation and a test using Mackenzie's result).

5.11 CLM Theorem

Recently, a new exact theorem has been proven for elastic composite problems in 2-D. There can be general anisotropy, and any number of phases, with perfect bonding between phases [22, 32, 33]. For this manual, we will consider only isotropic phases. Assume that the i'th phase has bulk and shear moduli K_i and G_i, and the overall effective moduli are K and G. Now make a transformation in each phase of the following form: $(1/K_i)' = 1/K_i - C$, $(1/G_i)' = 1/G_i + C$, where C is an constant such that the new elastic moduli in each phase are still positive. Then the theorem says that the new effective moduli, K' and G', are given in terms of the old moduli by $(1/K)' = 1/K - C$, and $(1/G)' = 1/G + C$. This theorem can then be used to test elastic programs like ELAS2D.F.

The same microstructure can be used as in the Hashin-Rosen example, Sec. 5.9. The phase moduli were originally $K_1 = 1$, $G_1 = 0.5$, $K_2 = 3$, $G_2 = 1$. The values of the effective moduli using these phase moduli are: $K = 1.6627$, $G = 0.711$, obtained using the program ELAS2D.F. Using a value of $C = 0.2$, the transformed phase moduli are $K_1 = 1.25$, $G_1 = 5/11$, $K_2 = 7.5$, $G_2 = 5/6$, and so the transformed effective moduli are expected to be: $K' = 2.492$, $G' = 0.6225$. The numerical values of the moduli, again obtained using ELAS2D.F, are: $K' = 2.499$, $G' = 0.6092$, an error of only 0.3% in K, and 2.1% in G. This error is a direct measure of how well pixels can resolve the true interface, since the CLM theorem assumes a perfect interface.

6 Other possible uses of programs

This section describes some different ways that the finite element and finite difference programs can be used. These uses are not, however, implemented in the programs discussed in this manual, but are meant to give ideas to potential users of these programs.

6.1 Fixed voltages and displacements

It is relatively easy in these programs to run a problem with a fixed displacement or a fixed voltage. To fix a given nodal variable at a certain value, simply give it that value initially, and then every time the gradient, gb, is calculated or changed, zero out the gradient with respect to this variable, so that it can never change. In the finite element programs, this needs to be done twice, once in ENERGY, after the gradient is first computed, and then also in DEMBX, after the gradient is updated using the Ah array, which results from the matrix multiply in DEMBX. In the finite difference programs, this zeroing out of gb is done only in DEMBX, since there is no ENERGY subroutine in the finite difference programs.

This technique can be used in THERMAL3D.F to be able to run an applied strain. One simply sets the initial values of the macrostrains to their desired values, and then zeroes out the gradient vector entries corresponding to these macrostrains. The macrostrains will then stay constant, and the solution will be obtained for an applied strain problem where eigenstrains can be specified.

For an electrical example of this process, see Ref. [11], where the intrinsic conductivity of superconducting centro-symmetric inclusions was computed, using the finite element method, by fixing the voltage on the nodes belonging to the inclusion and its surface. This procedure simulated a superconducting inclusion problem, in which the inclusion had no resistance and was therefore an equipotential body. In this problem, the voltage of the inclusion was set to zero at all the nodes composing the inclusion and its surface. A field was applied via periodic boundary conditions, and the rest of the voltages were then found that minimized the energy of the problem. The effective conductivity of the problem was found using the total energy, not the average current, since the current through the superconducting inclusion was not computed using this method. This was a case where the effective conductivity could only be defined via the energy.

An amusing electrical example where this technique can be used successfully is from an important problem in statistical physics, where it is desired to find the total resistance between two nodes in an infinite lattice. Each node is connected to its nearest neighbors by unit resistors. This problem has been worked out completely for the infinite 2-D square and the 3-D cubic lattice [34], analytically. Other papers that have looked at this problem numerically and analytically include [35, 36], which include tables of results.

The method of solution is usually to introduce a current source at the origin, and then an equal current sink at a lattice point of interest. The current is fixed, and the voltage between the two nodes is then calculated. The resistance between the two points is simply the ratio of voltage to current introduced, according to Ohm's law. The same problem can be done numerically, using the finite difference method, by fixing the voltages at the two nodes, computing the current in at the one node and out at the other node (which will be equal and opposite), and then computing the effective resistance between the two points via the ratio of fixed voltage difference to computed current. The conductivity chosen for the material being used is arbitrary, since the effective resistance will be proportional to the resistance of a single pixel. We compare to the analytical result for the 2-D square lattice and the 3-D simple cubic lattice, for two nodes separated by 20 lattice spacings, where all results are scaled by the resistance of a single pixel. In 2-D, the exact result for an infinite square lattice is 1.468191. The numerical result, for a 200×200 resistor network with periodic boundary conditions, is 1.473655, an error of only 0.4%. The exact result for an infinite 3-D simple cubic lattice is 0.497499, and the numerical result is 0.497905, for a $200 \times 200 \times 60$ lattice. The error in this case is only 0.1%.

Formulas and tables of numerical results for checking other spacings can be found in Refs. [34, 35, 36]. The integrals in Ref. [34] for the exact solutions can be easily evaluated, with high accuracy, using Gaussian quadratures. A program for generating the Gaussian points and weights for an N-point Gaussian quadrature evaluation is stored with the various programs described in this manual, and is called GAUSS.F. Recall that Gaussian quadrature is only for integrals between -1 and 1. A general integral needs to be changed into this form before using Gaussian quadrature [37]. A listing of GAUSS.F can be found in Sec. 9.3.6.

It is interesting to observe how the 2-D result for the resistance gradually becomes equal to the 3-D result, as the lattice grows in size in the vertical dimension. Figure 12 shows the resistance between nodes that are 20 lattice spacings apart, for lattices that were 200×200 in the horizontal direction, and from 0 (2-D) to 60 spacings thick in the vertical direction. By the time that the lattice is about as thick as the spacing between the two nodes of interest, the effective resistance has converged to within a few percent of the 3-D result.

Another application that arises in electrical problems is that of a piece of metal, usually of some odd shape, buried in or attached to a piece of material with finite conductivity. A field is applied from electrodes at either end of the sample, and one wishes to know the resulting voltage of the buried piece of metal. The electrodes have fixed voltages. The voltage must be the same throughout the metallic inclusion, but its value is free to float according to its shape, size, and position in order to solve the problem. One physical example would be an impedance spectroscopy experiment, where the current is applied via a working and counter electrode, and one wishes to know the voltage of a reference electrode that has been attached to or buried in the sample [38]. One way to handle this situation is to make a list of which nodes need to be at the same (but floating) voltage. The simplest starting point is to make all these voltages equal to zero. The conductances between the nodes in the inclusion can be made equal to one, for simplicity, though they will not really appear in the answer. Any positive number will suffice. Then, in every conjugate step where the gradient is computed, the gradient for all this list of nodes should be made equal to the average of all the individual gradients. In this way, the voltage of all these nodes will be forced to be always the same, simulating a metallic inclusion, but can gradually change in order to match the fields outside the inclusion. The correct solution will eventually be reached in the relaxation process.

6.2 Removing periodic boundary conditions

There will be times when one might not want to use the periodic boundary conditions that are built into the programs. It is simple to get rid of them. Simply use a unit cell that is $(nx + 1) \times (ny + 1) \times (nz + 1)$, and set the $i = nx + 1$, $j = ny + 1$, and $k = nz + 1$ pixels

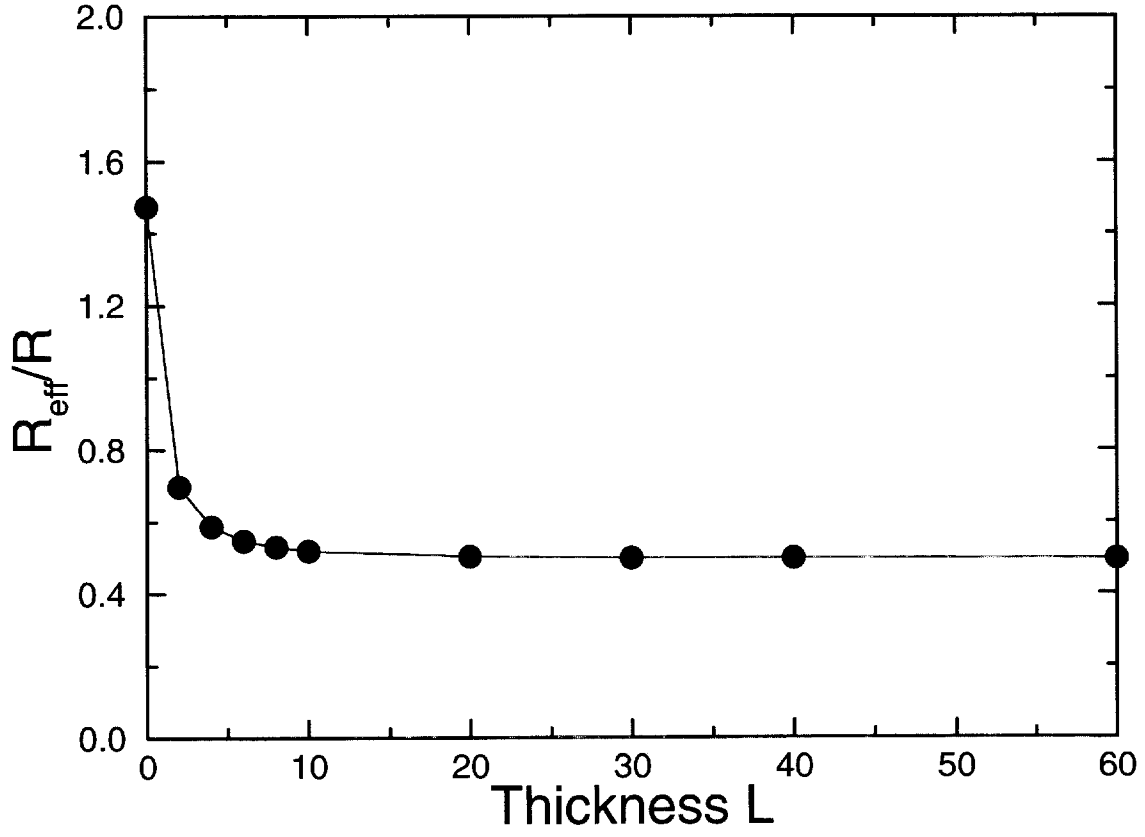


Figure 12: Showing the resistance between two nodes, separated by 20 lattice spacings, of a simple cubic network, normalized by the resistance of one bond, as a function of the thickness of the network.

to have zero conductivity or modulus, i.e. "air." In the finite element programs, this will effectively zero out the linear and quadratic term in the energy, which arise from the periodic boundary conditions. Terms that connect variables across the system will be multiplied by zero and thus contribute nothing. The program can then be run as is, but of course one must allow for the layer of air on the outside in the field averages. Also, one must put some kind of constraint on the boundary in order to have a non- zero result, since the applied electric field or strain came from the boundary conditions. One easy way to apply a uniform strain in an elastic problem is to use THERMAL3D.F, and allow every pixel to have the same eigenstrain. The system will then expand, but there will be internal relaxations because of the different moduli between phases. This is equivalent to applying an external strain [7]. One can also fix the displacements of the outermost pixels, to mimic the desired applied strain. In electrical problems, one can fix the voltage at one or more places. Other ways include putting a fixed force or current somewhere in the system, which is discussed next.

Usually when applying these programs to a real 2-D image, it will be desirable to remove periodic boundary conditions. Also, one will want to change an image, usually in *gif* or *tif* or *jpeg* or some other graphics format, into an image of phase labels. The hardest part is to convert the graphic image into an image of ASCII numbers. These number can then be easily read and converted into the appropriate phase labels, either outside the programs or within subroutine PPIXEL. The shareware program XV, easily obtainable on the Internet, has an output file format called *pgm* or *ppm*. This is an ASCII format of either gray scales from 0 to 255, or RGB triples, each ranging from 0 to 255. Simply read the image with XV, then output an ASCII file, which is then easily converted to phase labels using a separate small program.

6.3 Fixed currents and forces

Since the current in a resistor is just the conductance times the voltage difference between the two ends, it is somewhat easier to illustrate how to use fixed currents in the finite difference algorithm. Consider the problem discussed in Section 6.1, that of the effective resistance between two nodes of a lattice, in 2-D or 3-D. There the problem was solved by fixing the voltage difference between the two nodes, and then computing the current in and out the two nodes. The same problem can also be done by fixing the current in and out at the two nodes. The gradient of the energy at a node is just the negative of the current at that node. If we add 1 and -1 to the gradients at the two nodes before relaxation occurs, this will be like forcing a fixed current at these nodes. The voltages will relax, so as to force the total gradient to become zero, but with the current at the nodes fixed at its original value, since there are no other current or field sources in the problem. The same solution to the 2-D problem, with some slight round-off difference, was found as in Section 6.1. The same technique could then be used if fixed current boundary conditions, instead of periodic fixed voltage conditions, were desired. The currents at the boundary could be fixed using this method.

In the finite element programs, the negative gradient of the elastic energy with respect to a nodal displacement is just the force at that node. In the same way, in the subroutine ENERGY, after the gradient is first computed, a fixed value can be added to the gradient at a particular node, to simulate a fixed force at that node. If the subroutine ENERGY is then run again, after LDEMB conjugate gradient cycles in DEMBX, this addition should be renewed before going back into the conjugate gradient routine.

Usually the effect of fixed stresses is considered, not the effect of fixed forces. In this case, if for example a pressure p is to be put on a surface, the way one does this is to consider each pixel face, and put a force of $p/4$ on each node of each face, in 3-D. The edge length of the pixels is one. That way, nodes that are shared by four pixel faces will end up with p, nodes that are only shared by two pixel faces will have $p/2$, and nodes that are only on one pixel face, will have $p/4$. This takes care of overcounting. Consider four (2×2) pixel faces in a plane, containing 9 nodes. The total area is four, if the pixels have unit length. If a force p was put on each node, then the total pressure on the face would be $9p/4$, not p. However, if the method suggested is used, then the middle node will have a force p, the four side nodes will each have a force $p/2$, and the four corner nodes will each have a force $p/4$, for a total pressure of $4p/4 = p$. If the pixels are in a 1×4 arrangement, then the four corner nodes will have a force $p/4$, and the six middle nodes will each have a force of $p/2$, for a total pressure of $4p/4 = p$.

This use of fixed pressures can be tested using eq. (76), Mackenzie's exact result for an isotropic porous solid filled with a fluid at pressure p. If we consider a porous material with a dilute concentration of spherical holes, we can combine eq. (64) for the bulk modulus of such a system with Mackenzie's result to get an exact prediction for the strain expected at a pressure p. Alternatively, we can directly compute, using ELAS3D.F, the effective bulk modulus of the porous solid. For a 25^3 unit cell, containing a single spherical hole of diameter 11 pixels, when the solid frame bulk modulus was 30.0825, the effective bulk modulus of the porous material was 27.3909 as computed by ELAS3D.F. When $p = 0.1$, the prediction from Mackenzie's result is $\epsilon = 1.08882 \times 10^{-4}$. The numerical result, using the algorithm described above, was $\epsilon = 1.08888 \times 10^{-4}$, a difference of only 0.006%. So this algorithm works very well.

One warning: often pores will go across the boundary when periodic boundary conditions are used. In this case, this algorithm does not seem to work very well [41]. An alternate method can be to enclose the porous solid with a "skin" of effective material with elastic properties such that the bulk modulus of the composite system is unchanged from that of the porous solid alone. This algorithm does work on interior pores. When a skin is placed around the material, all pores become interior pores.

6.4 Surface energies

If there is a surface energy associated with a material, and the material has a very large surface area, a change in that surface energy can cause a measurable expansion or shrinkage of the material. This has been seen in Vycor glass, at very low relative partial pressures of an absorbed gas [39, 40]. As the partial pressure of the absorbed gas increases, the layer of gas molecules on the internal surface of the Vycor increases in thickness, and thus lowers the specific surface free energy. This lowering in surface energy allows an expansion to take place, which trades volume elastic energy for surface free energy. This expansion is linear in the change in surface free energy [41, 39, 40].

If a 3-D model of the material is available in a digital form, then a surface energy can be applied in finite element form. All surfaces are pixel faces, and are initially flat. One then must write, for general small strains of the pixel face, what the new area is in terms of the nodal displacements. One can then linearize this when the surface distortion is small (usually the overall strains are a percent or much less), to give an energy that can be added to the elastic energy, which is linear in the displacements.

The key to deriving a surface energy, γS_A, where S_A is the surface area of the digital image found by counting pixel faces and γ is the specific surface free energy, is knowing that each pixel surface is originally flat, before any strains are generated. Consider the $z = 0$ face ($1 - 2 - 3 - 4$ face) of a pixel labelled like that shown in Fig. 1. The coordinates of the number 1 node are $(x_1 + u_1, y_1 + v_1, z_1 + w_1)$, where (x_1, y_1, z_1) are the coordinates before any strain has occurred, and(u_1, v_1, w_1) are the x, y, and z displacements. There are similar formulas for the other nodes. No matter how the pixel face has been strained (assuming small distortions), the area of the face can be defined as the area of the triangle $1 - 2 - 3$, and the triangle $2 - 3 - 4$, since any three points are co-planar. The area of these two triangles can be written as one half the sum of the magnitude of the cross-products of the vectors making up their sides. The resulting formula contains the square root of the squares of various combinations of the differences of the nodal coordinates. The formula is somewhat simplified for this face by remembering that $z_1 = z_2 = z_3 = z_4 = 0$. Making the assumption of small differences between nodal displacements (small strains), the area of this face can be reduced to a linear form,

$$S_A = r^2[1 + \frac{1}{2r}(u_2 + u_3 + v_3 + v_4 - u_1 - u_4 - v_1 - v_2)] \tag{77}$$

where the pixel was originally an $r \times r \times r$ cube. For the $1-4-5-8$ face (x=0), the formula becomes

$$S_A = r^2[1 + \frac{1}{2r}(v_4 + v_8 + w_5 + w_8 - v_1 - v_5 - w_1 - w_4)] \tag{78}$$

and for the $1 - 2 - 6 - 5$ face ($y = 0$) the linearized formula is

$$S_A = r^2[1 + \frac{1}{2r}(u_2 + u_6 + w_5 + w_6 - u_1 - u_5 - w_1 - w_2)] \tag{79}$$

Consider a strain of 0.1 in the x-direction only. If the pixel has node 1 at the origin, then $u_1 = u_4 = u_5 = u_8 = 0$, $u_2 = u_3 = u_6 = u_7 = 0.1r$, and all the other displacement components are zero. Computing the above equations with these displacements gives $S_A = 1.1r^2$ for the $z = 0$ face, $1.1r^2$ for the $y = 0$ face, and $S_A = r^2$ for the $x = 0$ face, as expected.

A global vector can be built out of these displacements, which will also contain terms for the macrostrains, via the periodic boundary conditions, which are picked up at the boundaries. This vector is constant with respect to the macrostrains, and so will simply add to the gradient vector, and will not come into the subroutine DEMBX at all, since only the second derivative of the energy is used in that subroutine.

A simple example will suffice to show the accuracy of this technique [41]. Consider a solid block of material, with a dilute volume fraction c of spherical holes of radius R. In the dilute limit the holes can be treated separately, so the analytical problem can be carried out for a single hole. The solid material has bulk modulus K and shear modulus G, and is at zero strain. Now suddenly add a surface energy per unit area γ to the surface of the hole. The material will shrink in order to reduce the surface area of the hole and thus the surface energy stored. The reduction in surface energy from the shrinkage will be exactly balanced by the increase in volume stored elastic energy, giving the result:

$$\epsilon = \frac{-\gamma}{6R}\left(\frac{3}{G}+\frac{4}{K}\right)c \tag{80}$$

For a 15 pixel diameter spherial hole in a 40^3 unit cell, the agreement with the exact result for the coefficient of c in eq. (78) was 4.7% [41]. Better agreement would be obtained using a larger system.

7 Making and analyzing images and histograms

7.1 General features

It is often useful to make a current or stress map of the system being analyzed. These are helpful to check against exact solutions, and can often give insight into systems not capable of being understood analytically. The sections below will discuss current maps for the electrical programs, both finite difference and finite element, and stress maps for the finite element programs. Histograms are a way of making plots of the stress or current distribution functions. These distribution functions and their various moments are way of going one step beyond just analyzing the effective properties, and can give additional insight into the random system being studied. Also, the programs BURN2D.F and BURN3D.F will be discussed, which check an image for phase percolation.

7.2 Finite element electrical problems

It is simple to make a current map using the finite element programs ELECFEM2D.F or ELECFEM3D.F. In the subroutine CURRENT, the total average current is computed. In a small DO loop in the middle of the main DO loop, called DO 465, the average current for a given pixel is computed. The components of the average current in a pixel are called $cur1$, $cur2$, and $cur3$, for the x, y, and z directions. These variables can be written to a file, or perhaps only the magnitude of the current need be stored. Then an image, using some suitable scaling system (0-255 for gray scale, or a color system) can created using almost any kind of imaging software. The shareware program XV can also be used to convert the ASCII gray scale values back to a graphics format image.

In the finite element programs, the average current in a pixel is given in terms of the nodal voltages, using the average over the pixel of the conductivity tensor times the matrix in eq. (4). This matrix gives the local field in the pixel in terms of the nodal voltages, so the conductivity tensor times the local field gives the local current.

Figure 13 shows (top images), for the applied field in the horizontal direction, the horizontal current density, for two choices of the inclusion conductivity. The matrix always had a conductivity of 1, and the inclusion had either a conductivity of 10 (left) or 0.1 (right). The actual current densities have been scaled from 0 to 255 by assigning the value of 200 to the average current density. Any current density that scaled to a value over 255 was simply set to 255. Then a color scale was assigned according to this order: red, green, gray, and black (top to bottom, high to low current density in accompanying color bar). In the left top image of Fig. 13, the current tends to bunch up at the left and right of the inclusion, due to the normal current being continuous at the boundary. The current density is depleted at the top and bottom of the inclusion, as it is energetically favorable for the current to curve and go through the high conductivity inclusion.

The right top image of Fig. 13 shows a similar picture for an inclusion conductivity of 0.1. Notice now the current tends to bunch up at the top and bottom of the inclusion, as it is energetically more favorable to go around the low conductivity inclusion. Also, at the left and right of the inclusion, the current is low, because at the boundary the normal current must match the low current inside the inclusion.

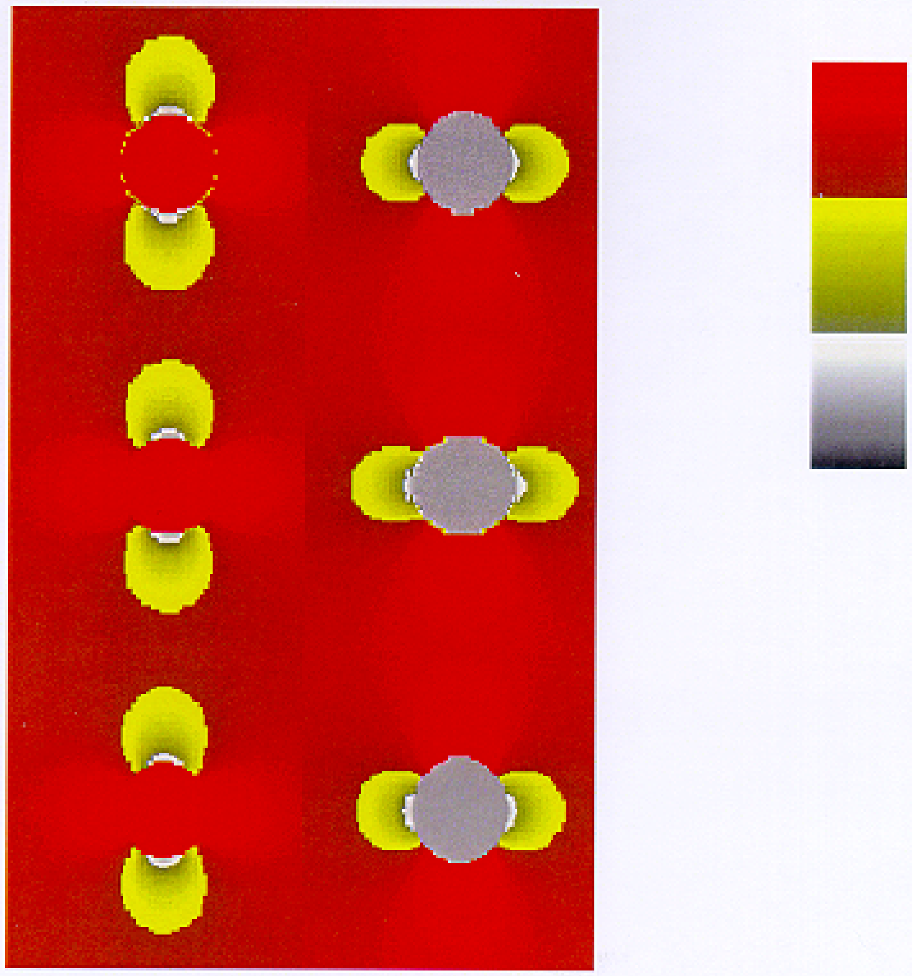


Figure 13: Image of the horizontal current magnitudes, with the applied electric field in the x-direction in all images. The inclusion is phase 2. Left: $\sigma_2 = 10$, right: $\sigma_2 = 0.1$, and both images had $\sigma_1 = 1.0$. Top: Finite element solution. Middle: finite difference solution. Bottom: exact solution, no periodic boundary conditions. Color bar shows high (red) to low (black) current scale.

In the top images of Fig. 13, one can notice a ring of miscolored pixels around the inclusion boundary. This is an artifact of having a digitally-rough approximation to a smooth boundary. Locally averaging the pixel intensities around the boundary would make the boundary much smoother, and give a better approximation to reality.

7.3 Finite difference electrical problems

In the finite difference electrical programs, one must be careful in defining the average current in a pixel. In d dimensions, there are $2d$ bonds coming into a node, with $2d$ currents to consider. The most obvious way to define the average current in a pixel is to average the current in the two x-bonds, the two y-bonds, and the two z-bonds, and thus obtain the three components of the average current vector in the pixel. Subroutine CURRENT in the finite difference programs computes the total current for the whole image by summing over all the

BLANK PAGE

pixel currents. Variables *cur*1, *cur*2, and *cur*3 are the local average currents in a pixel.

The middle images of Figure 13 shows the same problems as described in the previous section but now for a finite difference solution. The current maps are very similar, with similar small anomalies at the inclusion boundary. To the eye, there is very little difference between the finite difference and the finite element current maps. Recall from Fig. 8 and Table 10 that in this range of inclusion to matrix conductivity ratios, 0.1 to 10, the finite element and finite difference intrinsic conductivities agreed rather well.

The bottom images of Fig. 13 show the equivalent maps for an exact solution to the same problem, without periodic boundary conditions. The exact solution is described below, in Section 7.6. Most of the variation of the current is near to the inclusion, so that it is reasonable to compare infinite matrix analytical and periodic boundary numerical solutions. The finite difference and finite element solutions are seen, by comparing the different parts of Fig. 13, to at least qualitatively give accurate solutions of the conductivity problem. The analysis of the fields via their distribution in histogram form will show better the small differences between the exact and numerical solutions.

7.4 Finite element elastic problems

The subroutine STRESS, in the DO 465 loop, for all the elastic programs, computes the stress per pixel and adds these up to get the overall average stress. Simply output the variables *str*11, *str*22, *str*33, *str*13, *str*23, and *str*12, which are the local average stresses in a given pixel, to generate any kind of stress map. The local strains are also computed, *s*11, *s*22, *s*33, *s*13, *s*23, and *s*12, and can be output as well. In the thermal strain programs, THERMAL2D.F and THERMAL3D.F, the local strain and the local stress can be similarly output with or without the thermal strain included, if that is of interest in a given problem solution.

7.5 General features of histograms

When a microstructure is analyzed, then all the local currents/stresses/strains are available, since the full problem has been solved for every pixel. This information can be used to generate pictures or maps of what a quantity looks like throughout a microstructure. This is useful qualitative information. The same information can be used to generate histograms, or distribution functions, of a quantity of interest, which is a more rigorous way of looking at the same data. A histogram can give information on current or stress distributions and moments of these distributions. The basic kind of graph has area or volume fraction as the ordinate, and current or stress as the abscissa. A point with coordinates (g, p) means that a fraction p of the system has property g.

It should be noted that system averages of current or stress, that define the effective conductivity or elastic moduli, are generally quite reliable and accurate, as has been seen in the examples in Section 5. However, the actual pixel-by-pixel values of current or stress can be somewhat off, due mostly to discretization errors and digital boundaries. At a boundary which should be curved, locally one cannot have anything but flat pixel boundaries. This can throw off some of the values near the boundary.

7.6 Examples of histograms

Consider the same 200×200 system, with a 41-pixel diameter circular inclusion (phase 2) in the middle of the matrix (phase 1), as was used in Fig. 13. Histograms for the average current per pixel will be generated using both finite difference and finite element methods, and will be compared to the exact result, which uses the analytical solution of the problem.

This analytical solution is simple to generate. Using polar coordinates, (r, θ), the potential outside the circle is $-Er\cos\theta + B\cos\theta/r$, and the potential inside the circle is $-Ar\cos\theta$, where A and B are unknown constants and the applied field E is in the x-direction only. By requiring that the potential and the normal current be continuous at the $r = R$ boundary (the exterior of the circle), the coefficients A and B are determined via these two equations. The x component of the current is then:

$$\textit{Outside} \quad j_x = \sigma_1 E\left[1 + \frac{R^2(\sigma_2 - \sigma_1)(x^2 - y^2)}{(\sigma_1 + \sigma_2)(x^2 + y^2)^2}\right] \tag{81}$$

$$\textit{Inside} \quad j_x = \frac{2\sigma_1\sigma_2 E}{(\sigma_1 + \sigma_2)} \tag{82}$$

One problem in comparing the exact solution to the numerical solutions is that the exact solution assumes an infinite matrix, while the numerical solutions use periodic boundary conditions. For purposes of comparison, the currents from the exact solution at the pixel centers, constricted to a 200×200 area around the inclusion, will be used to create the exact solution histogram. An electric field of magnitude unity was applied in the x-direction. For all three computations, there will be $40,000$ numbers, the magnitude of the current in the x- direction for each pixel. The same range of bins, and bin sizes, were chosen for all three (finite element, finite difference, exact) so as to make the histograms more comparable. The exact solution could of course have been used to generate a histogram with infinite resolution.

In the first example, the isotropic conductivities were: $\sigma_1 = 1$, $\sigma_2 = 10$. Figure 14 shows all three histograms plotted from a minimum current of 0.2 to a maximum current of 2.0. The finite element and finite difference solutions actually had a few $(10-20)$ pixels that had higher currents than this, but as they would not show up on a graph, they have not been plotted. These come from digital boundaries, and are not relevant to such a comparison. The finite difference and finite element histograms are essentially the same, and are slightly different from the exact result. Part of this is due to the digital boundary of the circular inclusion, which leads to inaccuracies in the currents near the surface. Part of the disagreement is also due to the fact that the computations used periodic boundary conditions, while the exact solution is for an infinite matrix.

In the second example, the isotropic conductivities were $\sigma_1 = 1$ and $\sigma_2 = 0.1$. Figure 15 shows all three histograms plotted from a minimum current of 0.1 to a maximum current of 2.0. All currents were within these bounds. The finite difference and finite element histograms are essentially the same, and are slightly different from the exact result. Again, part of this is due to the digital boundary of the circular inclusion, which leads to inaccuracies in the currents near the surface. Similarly to Fig. 14, part of the disagreement between the analytical and numerical histograms is also due to the fact that the computations used periodic boundary conditions, while the exact solution was for an infinite matrix.

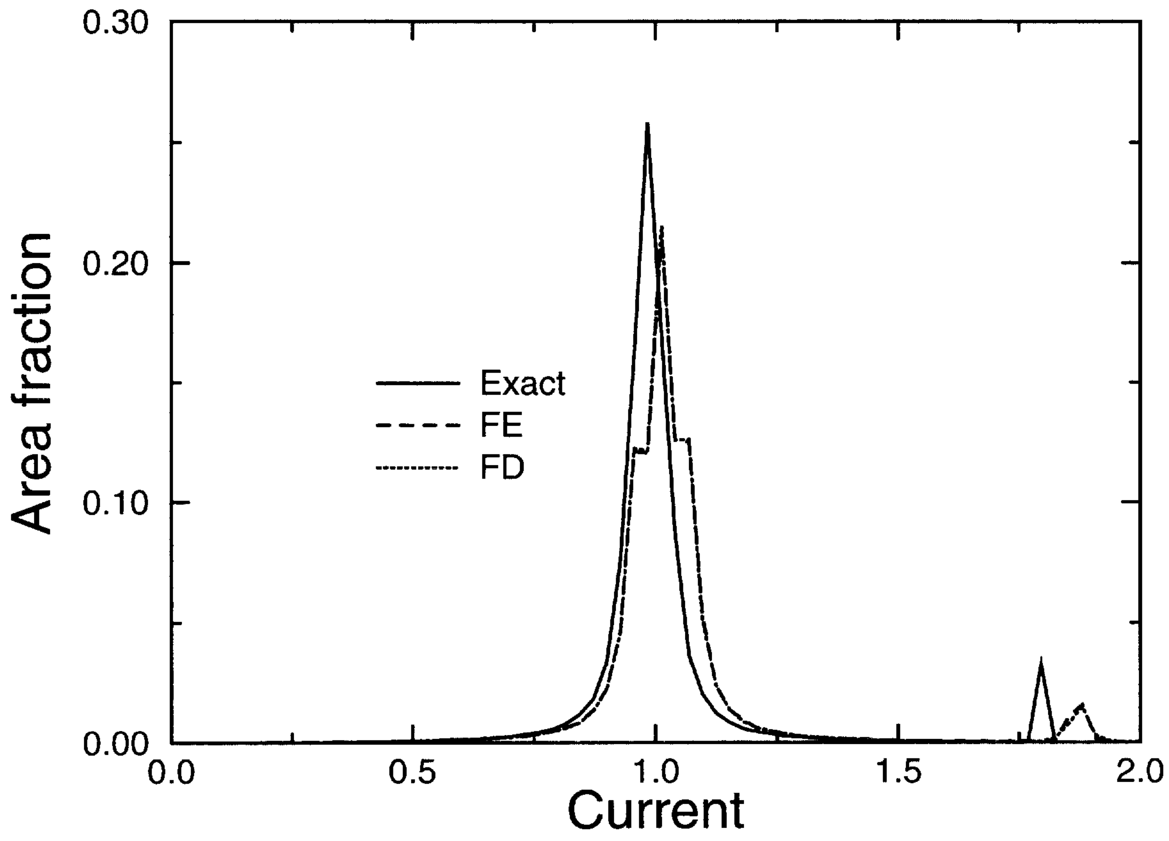


Figure 14: Current distribution for the same circular inclusion problem as in Fig. 13, calculated by all three methods (finite element, finite difference, and exact calculation, $\sigma_1 = 1.0$, $\sigma_2 = 10$.

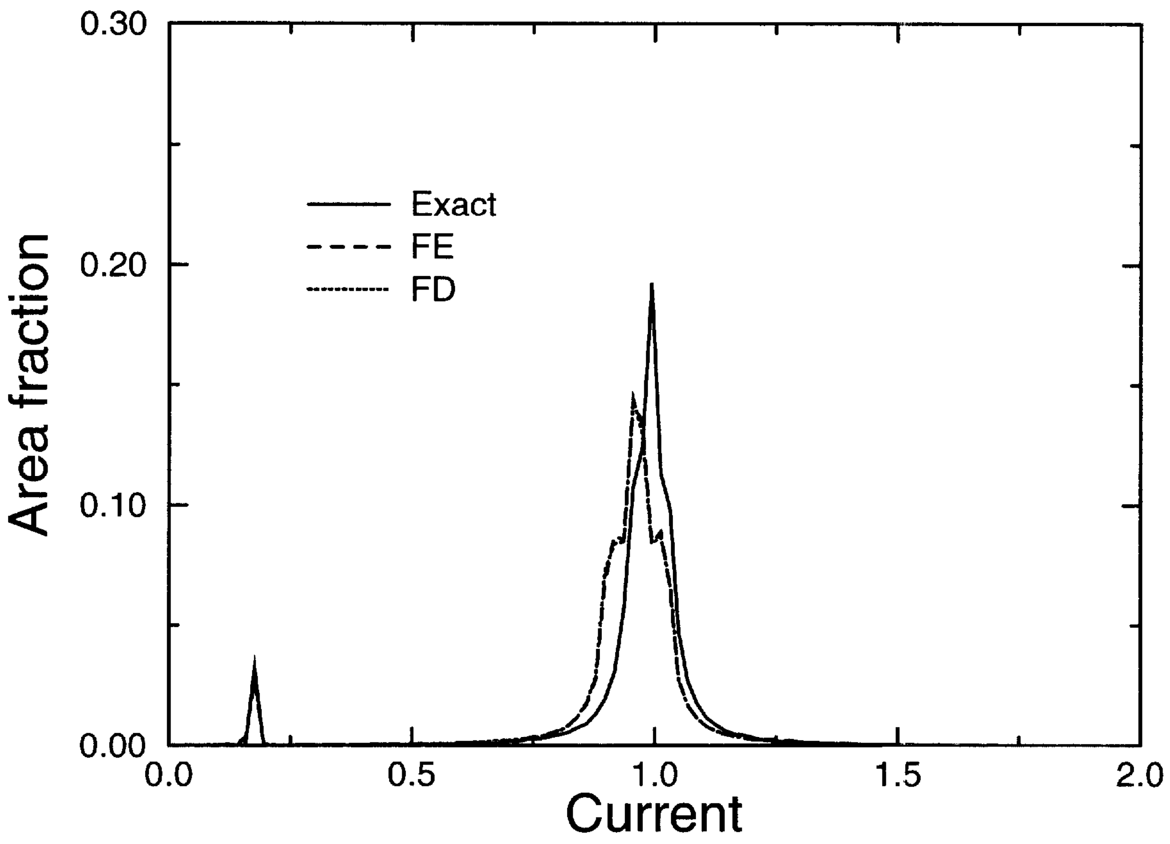


Figure 15: Current distribution for the same circular inclusion problem as in Fig. 13, calculated by all three methods (finite element, finite difference, and exact calculation, $\sigma_1 = 1.0$, $\sigma_2 = 0.1$

The results for Figs. 14 and 15 make it clear that current distributions can differ, but in such a way that averages (first order moments) are nearly the same. However, higher moments will differ more. This is why the error on the effective conductivity (1st moment) was always less than the error in the average field squared (2nd moment) in the checkerboard example given in Table 8.

7.7 Phase percolation in images

If a phase percolates in a given direction in a microstructure, that means that the phase is continuous from one side of the microstructure to the other in that direction. This is important topological information and affects greatly the ability of the phase to affect the overall properties [9, 10, 24, 42]. There are two auxiliary programs included with the finite element and finite difference programs, BURN2D.F and BURN3D.F, for computing phase percolation in 2-D and 3-D using the burning algorithm [42]. Given a phase label, these programs check for continuity of this phase in each of the principal directions.

The burning algorithm in principle "lights" a fire at one end of the microstructure, in the chosen phase, and lets the fire burn in that phase until there are no more pixels of that phase left unburned, at least ones that the fire can get to via nearest-neighbor connections. The other side of the microstructure is then checked to see if the fire reached there or not. If it did, then the chosen phase must be connected from one side to the other. If it did not, then the phase does not percolate.

The microstructure is read into the programs. As these are simple programs, the comments contained in the programs should suffice to describe their operation. These programs can be used to analyze an original microstructure. They can also be used to analyze a current or stress map, in the following way. Suppose we would like to know if the high currents in a current map are limited to a few hot spots, or are continuous across the microstructure. One could create an image where all pixels with currents above a certain value are given a phase label of one, and all other pixels a phase label of two. Then the burning programs could be used to see if the high current phase percolated or not.

The listing for BURN3D.F is given in Sec. 9.3.7. BURN2D.F is similar to the 3-D version.

8 References

[1] J.C. Maxwell, *A Treatise on Electricity and Magnetism* (Dover, New York, 1954).

[2] B.P. Flannery, H.W. Deckman, W.G. Roberge, and K.L. D'Amico, Science **235**, 1439 (1987).

[3] D.P. Bentz, N.S. Martys, P. Stutzman, M.S. Levenson, E.J. Garboczi, and L.M. Schwartz, "X-ray microtomography of an ASTM C-109 mortar exposed to sulfate attack," in *Microstructure of Cement-Based Systems/Bonding and Interfaces in Cementitious Materials*, edited by S. Diamond et al. (Materials Research Society Vol. 370, Pittsburgh, 1995), pp. 77-82.

[4] E.J. Garboczi and D.P. Bentz, "Fundamental Computer Simulation Models for Cement-Based Materials," in *Materials Science of Concrete* Vol. 2, edited by J. Skalny (American Ceramic Society, Westerville, Ohio, 1991).

[5] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, *Numerical Recipes* (Cambridge University Press, Cambridge, 1990).

[6] S. Feng, M.F. Thorpe, and E.J. Garboczi, Physical Review B **31**, 276-283 (1985).

[7] T. Mura, *Micromechanics of defects in solids: 2nd edition* (Martinus Nijhoff, Hingham, MA, 1987).

[8] J. Poutet, D. Manzoni, F. Hage-Chehade, C.J. Jacquin, M.J. Bouoteca, J.-F. Thovert, and P.M. Adler, "The effective mechanical properties of reconstructed porous media," Int. J. Rock Mech. Min. Sci. and Geomech. Abstr. **33**, 409-415 (1996); ibid, "The effective mechanical properties of random porous media," J. Mech. Phys. Solids **44**, 1587 (1996).

[9] E.J. Garboczi, M.F. Thorpe, M.S. DeVries, and A.R. Day, "Universal conductivity curve," Phys. Rev. A **43**, 6473 (1991).

[10] S. Kirkpatrick, "Percolation and conduction," Reviews of Modern Physics **45**, 574-588 (1971).

[11] J.F. Douglas and E.J. Garboczi, "Intrinsic viscosity and the polarizability of particles having a wide range of shapes," in Advances in Chemical Physics **16**, edited by I. Prigogine and S.A. Rice (John Wiley and Sons, New York, 1995).

[12] E.J. Garboczi and J.F. Douglas, "Intrinsic conductivity of objects having arbitrary shape and conductivity," Physical Review E **53**, 6169-6180 (1996).

[13] E.J. Garboczi and D.P. Bentz, "Analytical formulas for interfacial transition zone properties," Journal of Advanced Cement-Based Materials **6**, 99-108 (1997).

[14] S. Torquato, Applied Mechanics Review **44**, 37-76 (1991).

[15] Z. Hashin, "Analysis of composite materials: A survey," Journal of Applied Mechanics **50**, 481-505 (1983).

[16] W.F. Brown, Journal of Chemical Physics **23**, 1514-1517 (1955).

[17] S. Torquato, "Effective stiffness tensor of composite media-I. Exact series expansions," J. Mech. Phys. Solids **45**, 1421-1448 (1997).

[18] J.B. Keller, J. Math. Phys. **5**, 548 (1964).

[19] K.S. Mendelson, J. Appl. Phys. **46**, 917 (1975).

[20] M. Bobeth and G. Diener, "Field fluctuations in multi-component mixtures," J. Mech. Phys. Solids **34**, 1-17 (1986).

[21] R. Hill, Journal of the Mechanics and Physics of Solids **11**, 357-372 (1963).

[22] M.F. Thorpe and I. Jasiuk, Proc. Roy. Soc. London A **438**, 531-544 (1994).

[23] R.M. Christensen, *Mechanics of Composite Materials* (Krieger Publishing Co., Malabar, FL, 1991).

[24] K.A. Snyder, E.J. Garboczi, and A.R. Day, "The elastic moduli of simple two-dimensional isotropic composites: Computer simulation and effective medium theory," J. Appl. Phys. **72**, 5948-5955 (1992).

[25] L. Eyges and P. Gianino, IEEE Transactions on Antennas and Prop. **27**, 557 (1979).

[26] L. Vegard, Z. Phys. **5**, 17 (1921); M.F. Thorpe and E.J. Garboczi, Phys. Rev. B **42**, 8405-8417 (1990).

[27] J.N. Goodier, Phil. Mag. Series 7, **23**, 1017-1032 (1937).

[28] T. Mura, H.M. Shodja, T.Y. Lin, A. Safadi, and K. Hirashima, "The determination of the elastic field of a star shaped inclusion," presented at the Twelfth U.S. National Congress of Applied Mechanics, University of Washington, Seattle, 1994.

[29] J. Wulf, P. Lipetzky, G.E. Beltz, and T. Steinkopff, "A finite element model of the stress field in a star-shaped inclusion," Comp. Mater. Sci. **3**, 423-429 (1995).

[30] B.W. Rosen and Z. Hashin, "Effective thermal expansion coefficient and specific heats of composite materials," International Journal of Engineering Science **8**, 157-173 (1970).

[31] J.K. Mackenzie, "The elastic constants of a solid containing spherical holes," Proc. Roy. Soc. **683**, 2-11 (1950).

[32] A. Cherkaev, K. Lurie, and G.W. Milton, "Invariant properties of the stress in plane elasticity and equivalence classes of composites," Proc. Roy. Soc. Lond. A **438**, 519-529 (1992).

[33] K.A. Lurie and A.V. Cherkaev, "The effective properties of composite materials and problems of optimal designs of constructions," Usp. Mekaniki (Adv. Mech.) **9**, 3-81 (1986).

[34] H. Davies, "Poisson's partial difference equation," Quart. J. Math. Oxford **6**, 232-40 (1955).

[35] G. Venezian, "On the resistance between two points on a grid," Am. J. Phys. **62**, 1000-1004 (1994). Note that in eq. (21), there is a typographical error. The constant "0.51469" should be "3.23386."

[36] G. Montet, "The effective resistance of passive networks," J. Math. Phys. **5**, 1555-1559 (1964).

[37] M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions* (Dover, New York, 1965), pp. 887-888.

[38] G. Hsieh, T.O. Mason, E.J. Garboczi, and L.R. Pederson, "Experimental limitations in impedance spectroscopy: Part III–Effect of reference electrode geometry/position," Solid State Ionics **96**, 153-172 (1997).

[39] D.J.C. Yates, "The expansion of porous glass on the adsorption of non-polar gases," Proc. Roy. Soc. London A **224**, 526-544 (1954).

[40] G.W. Scherer, "Dilatation of porous glass," Journal of the American Ceramic Society **69**, 473-480 (1986).

[41] D.P. Bentz, E.J. Garboczi, and D.A. Quenard, "Modelling drying shrinkage in reconstructed porous materials: Application to porous Vycor glass," Mod. Sim. Mater. Sci. Eng. **6**, 211-236 (1998).

[42] D. Stauffer and A. Aharony, *Introduction to percolation theory 2nd Ed.* (Taylor and Francis, London, 1992).

9 Listing of programs and computer requirements

9.1 Computer details, access information

The programs discussed in this manual were all written in FORTRAN 77, and use no special statements that are machine specific. The programs themselves are straightforward computational algorithms. In fact, these programs could quite easily be translated into another language, or used by someone who had no previous knowledge of FORTRAN, just by learning the meaning of a few statements that are analogous to those in C, for example. All programs should be run in double precision (8 bytes or 64 bits per real variable). Running the finite difference routines in single precision may give reasonable answers, but double precision is more trustworthy. Running the finite element routines in single precision will not work, except perhaps for very small systems, on the order of $10^3 = 1000$ nodes.

In the finite element programs, all the main arrays are passed to subroutines via simple common statements. Small variables and parameters are passed in the subroutine calls. In contrast to this, in the finite difference programs, all variables, including the main arrays, are passed via subroutine call parameter lists. There are no common blocks used. In the finite element programs, the dimensions of the system can be easily changed by doing a global replace on the dimensions shown in the array declarations in the main program, since the dimensions of the main arrays in the subroutines must be changed as well. The values of nx, ny, and nz are then changed to match. The dimensions of the main arrays in the finite difference programs only need to be changed in the main program. There is no reason why the system studied has to be a cube. All the programs will handle a rectangular parallelipiped with $nx \neq ny \neq nz$. There are no input files except that for the system digital image. If the user desires to have the various phase conductivities, elastic moduli, etc. given in input files, that is easy to introduce into the programs.

The programs discussed in this manual are available via anonymous ftp from the directory *edsel/ftp/pub/FDFEMANUAL*, on the computer *edsel.cbt.nist.gov*. This manual is also available online, as in Chapter 2 in an HTML electronic monograph on the computer modelling of cement-based materials, at:

http://ciks.cbt.nist.gov/garboczi

See this address for instructions of how to down-load postscript and pdf versions of this manual. The manual may also be requested from the author in paper form:

edward.garboczi@nist.gov

9.2 Memory requirements

The memory requirements of the various programs can be expressed in terms of the bytes of memory required per pixel of the microstructure. The following numbers assume that double precision (REAL*8 and COMPLEX*16) is used. Only the memory requirements for arrays that are of the system size are counted. Variables like *dk*, *phasemod*, and *sigma* are ignored in this count, since their memory requirements scale with the number of phases, and this number is usually of order one.

Program	Bytes per pixel
DC2D.F	54
DC3D.F	62
AC2D.F	102
AC3D.F	118
ELECFEM2D.F	78
ELECFEM3D.F	150
ELAS2D.F	118
ELAS3D.F	230
THERMAL2D.F	134
THERMAL3D.F	254

Table 13: Memory requirements in terms of bytes per pixel for programs discussed in this manual.

9.3 Listing of key programs

The FORTRAN 77 code is listed below for the following main programs: ELECFEM3D.F (example of finite element electrical conductivity program), ELAS3D.F (example of finite element elastic program), THERMAL3D.F (example of finite element elastic program with thermal strains and variable system shape and size), DC3D.F (example of real finite difference electrical program), and AC3D.F (example of complex finite difference electrical program). The other programs are 2-D equivalents of these programs. Listings for two auxiliary programs, GAUSS.F (Gaussian quadrature weights and points) and BURN3D.F (phase percolation using burning algorithm), are also given.

9.3.1 ELECFEM3D.F

```
c  ************************  elecfem3d.f  ****************************
c  BACKGROUND

c  This program solves Laplace's equation in a random conducting
c  material using the finite element method.  Each pixel in the 3-D digital
c  image is a cubic tri-linear finite element,  having its own conductivity.
c  Periodic boundary conditions are maintained.  In the comments below,
c  (USER) means that this is a section of code that the user might
c  have to change for his particular problem.  Therefore the user is
c  encouraged to search for this string.

c  PROBLEM AND VARIABLE DEFINITION

c  The problem being solved is the minimization of the energy
c  1/2 uAu + b u + C, where A is the Hessian matrix composed of the
c  stiffness matrices (dk) for each pixel/element, b is a constant vector
c  and C is a constant that are determined by the applied field and
c  the periodic boundary conditions, and u is a vector of all the voltages.
c  The method used is the conjugate gradient relaxation algorithm.
c  Other variables are:  gb is the gradient = Au+b, h and Ah are
c  auxiliary variables used in the conjugate gradient algorithm (in dembx),
c  dk(n,i,j) is the stiffness matrix of the n'th phase, sigma(n,i,j) is
c  the conductivity tensor of the n'th phase, pix is a vector that gives
c  the phase label of each pixel, ib is a matrix that gives the labels of
c  the 27 (counting itself) neighbors of a given node, prob is the volume
c  fractions of the various phases, and currx, curry, currz are the
c  volume averaged total currents in the x, y, and z directions.

c DIMENSIONS

c  The vectors u,gb,b,h, and Ah are dimensioned to be the system size,
c  ns=nx*ny*nz, where the digital image of the microstructure considered
c  is a rectangular parallelipiped ( nx x ny x nz) in size.
c  The arrays pix and ib are also dimensioned to the system size.
c  The array ib has 27 components, for the 27 neighbors of a node.
c  Note that the program is set up at present to have at most 100
c  different phases.  This can easily be changed, simply by changing
c  the dimensions of dk, prob, and sigma. Nphase gives the number of
c  phases being considered.
c  All arrays are passed between subroutines using simple common statements.

c  STRONGLY SUGGESTED:  READ THE MANUAL BEFORE USING PROGRAM!!
```

```
c  (USER) Change these dimensions and in other subroutines at the same time.
c  For example, search and replace all occurrences throughout the program
c  of "(8000" by "(64000", to go from a 20 x 20 x 20 system to a
c  40 x 40 x 40 system.
real u(8000),gb(8000),b(8000),dk(100,8,8)
        real h(8000),Ah(8000)
real sigma(100,3,3),prob(100)
integer in(27),jn(27),kn(27)
integer*4 ib(8000,27)
        integer*2 pix(8000)

common/list1/currx,curry,currz
common/list2/ex,ey,ez
common/list3/ib
        common/list4/pix
common/list5/dk,b,C
common/list6/u
common/list7/gb
common/list8/sigma
common/list9/h,Ah

c  (USER) Unit 9 is the microstructure input file, unit 7 is
c  the results output file.
        open(9,file='microstructure.dat')
        open(7,file='outputfile.out')

c  (USER) nx,ny,nz give the size of the lattice
      nx=20
      ny=20
      nz=20
c ns=total number of sites
      ns=nx*ny*nz
      write(7,9010) nx,ny,nz,ns
9010  format('nx= ',i4,' ny= ',i4,' nz= ',i4,' ns = ',i8)

c  (USER) nphase is the number of phases being considered in the problem.
c  The values of pix(m) will run from 1 to nphase.
      nphase=2

c  (USER) gtest is the stopping criterion, compared to gg=gb*gb.
c  gtest=abc*ns, so that when gg < gtest, that average value per pixel
c  of gb is less than sqrt(abc).
      gtest=1.e-16*ns

c  Construct the neighbor table, ib(m,n)
```

```
c  First construct 27 neighbor table in terms of delta i, delta j, delta k
c  (See Table 3 in manual)
      in(1)=0
      in(2)=1
      in(3)=1
      in(4)=1
      in(5)=0
      in(6)=-1
      in(7)=-1
      in(8)=-1

      jn(1)=1
      jn(2)=1
      jn(3)=0
      jn(4)=-1
      jn(5)=-1
      jn(6)=-1
      jn(7)=0
      jn(8)=1

      do 555 n=1,8
      kn(n)=0
      kn(n+8)=-1
      kn(n+16)=1
      in(n+8)=in(n)
      in(n+16)=in(n)
      jn(n+8)=jn(n)
      jn(n+16)=jn(n)
555   continue
      in(25)=0
      in(26)=0
      in(27)=0
      jn(25)=0
      jn(26)=0
      jn(27)=0
      kn(25)=-1
      kn(26)=1
      kn(27)=0

c  Now construct neighbor table according to 1-d labels
c  Matrix ib(m,n) gives the 1-d label of the n'th neighbor (n=1,27) of
c  the node labelled m.
      nxy=nx*ny
      do 1020 k=1,nz
      do 1020 j=1,ny
      do 1020 i=1,nx
      m=nxy*(k-1)+nx*(j-1)+i
      do 1004 n=1,27
      i1=i+in(n)
      j1=j+jn(n)
      k1=k+kn(n)
      if(i1.lt.1) i1=i1+nx
      if(i1.gt.nx) i1=i1-nx
      if(j1.lt.1) j1=j1+ny
      if(j1.gt.ny) j1=j1-ny
      if(k1.lt.1) k1=k1+nz
      if(k1.gt.nz) k1=k1-nz
      m1=nxy*(k1-1)+nx*(j1-1)+i1
      ib(m,n)=m1
1004  continue
1020  continue

c  Compute the electrical conductivity of each microstructure.
c  (USER) npoints is the number of microstructures to use.
        npoints=1
        do 8000 micro=1,npoints

c  Read in a microstructure in subroutine ppixel, and set up pix(m)
c  with the appropriate phase assignments.
        call ppixel(nx,ny,nz,ns,nphase)
c Count and output the volume fractions of the different phases
        call assig(ns,nphase,prob)
do 805 i=1,nphase
write(7,*) 'Volume fraction of phase ',i,' = ',prob(i)
805     continue

c  (USER) sigma(100,3,3) is the electrical conductivity tensor of each phase
c  The user can make the value of sigma to be different for each
c  phase of the microstructure if so desired (up to 100 phases as currently
c  dimensioned).
      sigma(1,1,1)=1.0
      sigma(1,1,2)=0.0
      sigma(1,1,3)=0.0
      sigma(1,2,2)=1.0
      sigma(1,2,3)=0.0
      sigma(1,3,3)=1.0
      sigma(1,2,1)=sigma(1,1,2)
      sigma(1,3,1)=sigma(1,1,3)
      sigma(1,3,2)=sigma(1,2,3)
      sigma(2,1,1)=0.5
      sigma(2,1,2)=0.0
      sigma(2,1,3)=0.0
      sigma(2,2,2)=0.5
      sigma(2,2,3)=0.0
      sigma(2,3,3)=0.5
      sigma(2,2,1)=sigma(2,1,2)
      sigma(2,3,1)=sigma(2,1,3)
      sigma(2,3,2)=sigma(2,2,3)

c  write out the phase electrical conductivity tensors
      do 11 i=1,nphase
      write(7,*) 'Phase ',i,' conductivity tensor is:'
      write(7,*) sigma(i,1,1),sigma(i,1,2),sigma(i,1,3)
      write(7,*) sigma(i,2,1),sigma(i,2,2),sigma(i,2,3)
      write(7,*) sigma(i,3,1),sigma(i,3,2),sigma(i,3,3)
11    continue

c  (USER) Set applied electric field.
          ex=1.0
          ey=1.0
          ez=1.0
      write(7,*) 'Applied field components:'
      write(7,*) 'ex = ',ex,' ey = ',ey,' ez = ',ez

c  Set up the finite element "stiffness" matrices and the Constant and
c  vector required for the energy

call femat(nx,ny,nz,ns,nphase)

c  Apply a homogeneous macroscopic electric field as the initial condition
do 1050 k=1,nz
do 1050 j=1,ny
        do 1050 i=1,nx
m=nxy*(k-1)+nx*(j-1)+i
x=float(i-1)
y=float(j-1)
z=float(k-1)
u(m)=-x*ex-y*ey-z*ez
1050 continue


c  Relaxation Loop
c  (USER) kmax is the maximum number of times dembx will be called, with
```

```
c  ldemb conjugate gradient steps done during each call. The total
c  number of conjugate gradient cycles allowed for a given conductivity
c  computation is kmax*ldemb.
         kmax=40
         ldemb=50
         ltot=0

c  Call energy to get initial energy and initial gradient
         call energy(nx,ny,nz,ns,utot)
c  gg is the norm squared of the gradient (gg=gb*gb)
   gg=0.0
         do 100 m=1,ns
         gg=gg+gb(m)*gb(m)
100      continue
write(7,*) 'Initial energy = ',utot,'gg = ',gg
         call flush(7)

         do 5000 kkk=1,kmax
c  Call dembx to go into conjugate gradient solver
         call dembx(ns,Lstep,gg,dk,gtest,ldemb,kkk)
         ltot=ltot+Lstep
c  Call energy to compute energy after dembx call. If gg < gtest, this
c  will be the final energy. If gg is still larger than gtest, then this
c  will give an intermediate energy with which to check how the relaxation
c  process is coming along.
         call energy(nx,ny,nz,ns,utot)
write(7,*) 'Energy = ',utot,'gg = ',gg
write(7,*) ltot, ' conj. grad. steps'
         if(gg.lt.gtest) goto 444

c  If relaxation process will continue, compute and output currents as an
c  additional aid to judge how the relaxation procedure if progressing.
        call current(nx,ny,nz,ns)
c Output intermediate currents
         write(7,*)
write(7, *) ' Current in x direction = ',currx
write(7, *) ' Current in y direction = ',curry
write(7, *) ' Current in z direction = ',currz
         call flush(7)

5000    continue

444     call current(nx,ny,nz,ns)

c Output final currents
```

```
        write(7,*)
write(7, *) ' Current in x direction = ',currx
write(7, *) ' Current in y direction = ',curry
write(7, *) ' Current in z direction = ',currz
        call flush(7)

8000    continue

        end

c  Subroutine that sets up the stiffness matrices, linear term in
c  voltages, and constant term C that appear in the total energy due
c  to the periodic boundary conditions.

      subroutine femat(nx,ny,nz,ns,nphase)
      real dk(100,8,8),xn(8),b(8000),C
      real dndx(8),dndy(8),dndz(8)
      real g(3,3,3),sigma(100,3,3)
      real es(3,8)
      integer is(8)
      integer*4 ib(8000,27)
      integer*2 pix(8000)

common/list2/ex,ey,ez
common/list3/ib
        common/list4/pix
common/list5/dk,b,C
common/list8/sigma

      nxy=nx*ny

c  initialize stiffness matrices
      do 40 m=1,nphase
      do 40 j=1,8
      do 40 i=1,8
      dk(m,i,j)=0.0
40    continue

c  set up Simpson's rule integration weight vector
      do 30 k=1,3
      do 30 j=1,3
      do 30 i=1,3
      nm=0
      if(i.eq.2) nm=nm+1
      if(j.eq.2) nm=nm+1
      if(k.eq.2) nm=nm+1
      g(i,j,k)=4.0**nm
30    continue

c  loop over the nphase kinds of pixels and Simpson's rule quadrature
c  points in order to compute the stiffness matrices.  Stiffness matrices
c  of trilinear finite elements are quadratic in x, y, and z, so that
c  Simpson's rule quadrature is exact.
      do 4000 ijk=1,nphase
      do 3000 k=1,3
      do 3000 j=1,3
      do 3000 i=1,3
      x=float(i-1)/2.0
      y=float(j-1)/2.0
      z=float(k-1)/2.0
c  dndx means the negative derivative with respect to x of the shape
c  matrix N (see manual, Sec. 2.2), dndy, dndz are similar.
      dndx(1)=-(1.0-y)*(1.0-z)
      dndx(2)=(1.0-y)*(1.0-z)
      dndx(3)=y*(1.0-z)
      dndx(4)=-y*(1.0-z)
      dndx(5)=-(1.0-y)*z
      dndx(6)=(1.0-y)*z
      dndx(7)=y*z
      dndx(8)=-y*z
      dndy(1)=-(1.0-x)*(1.0-z)
      dndy(2)=-x*(1.0-z)
      dndy(3)=x*(1.0-z)
      dndy(4)=(1.0-x)*(1.0-z)
      dndy(5)=-(1.0-x)*z
      dndy(6)=-x*z
      dndy(7)=x*z
      dndy(8)=(1.0-x)*z
      dndz(1)=-(1.0-x)*(1.0-y)
      dndz(2)=-x*(1.0-y)
      dndz(3)=-x*y
      dndz(4)=-(1.0-x)*y
      dndz(5)=(1.0-x)*(1.0-y)
      dndz(6)=x*(1.0-y)
      dndz(7)=x*y
      dndz(8)=(1.0-x)*y
c  now build electric field matrix
      do 2799 n1=1,3
      do 2799 n2=1,8
      es(n1,n2)=0.0
```

```
2799  continue
      do 2797 n=1,8
      es(1,n)=dndx(n)
      es(2,n)=dndy(n)
      es(3,n)=dndz(n)
2797  continue
c  now do matrix multiply to determine value at (x,y,z), multiply by
c  proper weight, and sum into dk, the stiffness matrix
      do 900 ii=1,8
      do 900 jj=1,8
c  Define sum over field matrices and conductivity tensor that defines
c  the stiffness matrix.
      sum=0.0
      do 890 kk=1,3
      do 890 ll=1,3
      sum=sum+es(kk,ii)*sigma(ijk,kk,ll)*es(ll,jj)
890   continue
      dk(ijk,ii,jj)=dk(ijk,ii,jj)+g(i,j,k)*sum/216.
900   continue

3000  continue
4000  continue

c  Set up vector for linear term, b, and constant term, C,
c  in the electrical energy.  This is done using the stiffness matrices,
c  and the periodic terms in the applied field that come in at the boundary
c  pixels via the periodic boundary conditions and the condition that
c  an applied macroscopic field exists (see Sec. 2.2 in manual).

      do 5000 m=1,ns
      b(m)=0.0
5000  continue

c  For all cases, correspondence between 1-8 finite element node labels
c  and 1-27 neighbor labels is:  1:ib(m,27),2:ib(m,3),3:ib(m,2),
c  4:ib(m,1),5:ib(m,26),6:ib(m,19),7:ib(m,18),8:ib(m,17)
c  (see Table 4 in manual)
      is(1)=27
      is(2)=3
      is(3)=2
      is(4)=1
      is(5)=26
      is(6)=19
      is(7)=18
      is(8)=17
      C=0.0
c  x=nx face
      i=nx
      do 2001 i8=1,8
      xn(i8)=0.0
      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
      xn(i8)=-ex*nx
      end if
2001  continue
      do 2000 j=1,ny-1
      do 2000 k=1,nz-1
      m=nxy*(k-1)+j*nx
      do 1900 mm=1,8
      sum=0.0
      do 1899 m8=1,8
      sum=sum+xn(m8)*dk(pix(m),m8,mm)
      C=C+0.5*xn(m8)*dk(pix(m),m8,mm)*xn(mm)
1899  continue
      b(ib(m,is(mm)))=b(ib(m,is(mm)))+sum
1900  continue
2000  continue
c  y=ny face
      j=ny
      do 2011 i8=1,8
      xn(i8)=0.0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      xn(i8)=-ey*ny
      end if
2011  continue
      do 2010 i=1,nx-1
      do 2010 k=1,nz-1
      m=nxy*(k-1)+nx*(ny-1)+i
      do 1901 mm=1,8
      sum=0.0
      do 2099 m8=1,8
      sum=sum+xn(m8)*dk(pix(m),m8,mm)
      C=C+0.5*xn(m8)*dk(pix(m),m8,mm)*xn(mm)
2099  continue
      b(ib(m,is(mm)))=b(ib(m,is(mm)))+sum
1901  continue
2010  continue
c  z=nz face
      k=nz
      do 2021 i8=1,8
      xn(i8)=0.0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      xn(i8)=-ez*nz
      end if
2021  continue
      do 2020 i=1,nx-1
      do 2020 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      do 1902 mm=1,8
      sum=0.0
      do 2019 m8=1,8
      sum=sum+xn(m8)*dk(pix(m),m8,mm)
      C=C+0.5*xn(m8)*dk(pix(m),m8,mm)*xn(mm)
2019  continue
      b(ib(m,is(mm)))=b(ib(m,is(mm)))+sum
1902  continue
2020  continue
c  x=nx y=ny edge
      i=nx
      y=ny
      do 2031 i8=1,8
      xn(i8)=0.0
      if(i8.eq.2.or.i8.eq.6) then
      xn(i8)=-ex*nx
      end if
      if(i8.eq.4.or.i8.eq.8) then
      xn(i8)=-ey*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      xn(i8)=-ey*ny-ex*nx
      end if
2031  continue
      do 2030 k=1,nz-1
      m=nxy*k
      do 1903 mm=1,8
      sum=0.0
      do 2029 m8=1,8
      sum=sum+xn(m8)*dk(pix(m),m8,mm)
      C=C+0.5*xn(m8)*dk(pix(m),m8,mm)*xn(mm)
2029  continue
      b(ib(m,is(mm)))=b(ib(m,is(mm)))+sum
1903  continue
2030  continue
c  x=nx z=nz edge
      i=nx
      k=nz
      do 2041 i8=1,8
      xn(i8)=0.0
      if(i8.eq.2.or.i8.eq.3) then
      xn(i8)=-ex*nx
      end if
      if(i8.eq.5.or.i8.eq.8) then
      xn(i8)=-ez*nz
      end if
      if(i8.eq.6.or.i8.eq.7) then
      xn(i8)=-ez*nz-ex*nx
      end if
2041  continue
      do 2040 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      do 1904 mm=1,8
      sum=0.0
      do 2039 m8=1,8
      sum=sum+xn(m8)*dk(pix(m),m8,mm)
      C=C+0.5*xn(m8)*dk(pix(m),m8,mm)*xn(mm)
2039  continue
      b(ib(m,is(mm)))=b(ib(m,is(mm)))+sum
1904  continue
2040  continue
c  y=ny z=nz edge
      j=ny
      k=nz
      do 2051 i8=1,8
      xn(i8)=0.0
      if(i8.eq.5.or.i8.eq.6) then
      xn(i8)=-ez*nz
      end if
      if(i8.eq.3.or.i8.eq.4) then
      xn(i8)=-ey*ny
      end if
      if(i8.eq.7.or.i8.eq.8) then
      xn(i8)=-ey*ny-ez*nz
      end if
2051  continue
      do 2050 i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      do 1905 mm=1,8
      sum=0.0
      do 2049 m8=1,8
      sum=sum+xn(m8)*dk(pix(m),m8,mm)
      C=C+0.5*xn(m8)*dk(pix(m),m8,mm)*xn(mm)
2049  continue
      b(ib(m,is(mm)))=b(ib(m,is(mm)))+sum
1905  continue
2050  continue
c  x=nx y=ny z=nz corner
      i=nx
      j=ny
      k=nz
      do 2061 i8=1,8
      xn(i8)=0.0
      if(i8.eq.2) then
      xn(i8)=-ex*nx
      end if
      if(i8.eq.4) then
      xn(i8)=-ey*ny
      end if
      if(i8.eq.5) then
      xn(i8)=-ez*nz
      end if
      if(i8.eq.8) then
      xn(i8)=-ey*ny-ez*nz
      end if
      if(i8.eq.6) then
      xn(i8)=-ex*nx-ez*nz
      end if
      if(i8.eq.3) then
      xn(i8)=-ex*nx-ey*ny
      end if
      if(i8.eq.7) then
      xn(i8)=-ex*nx-ey*ny-ez*nz
      end if
2061  continue
      m=nx*ny*nz
      do 1906 mm=1,8
      sum=0.0
      do 2059 m8=1,8
      sum=sum+xn(m8)*dk(pix(m),m8,mm)
      C=C+0.5*xn(m8)*dk(pix(m),m8,mm)*xn(mm)
2059  continue
      b(ib(m,is(mm)))=b(ib(m,is(mm)))+sum
1906  continue

      return
      end
```

```
c  Subroutine computes the total energy, utot, and gradient, gb

      subroutine energy(nx,ny,nz,ns,utot)
real u(8000),gb(8000)
real b(8000),C
real dk(100,8,8)
real utot
  integer*4 ib(8000,27)
        integer*2 pix(8000)

common/list2/ex,ey,ez
common/list3/ib
        common/list4/pix
common/list5/dk,b,C
common/list6/u
common/list7/gb

do 2090 m=1,ns
gb(m)=0.0
2090 continue

c Energy loop. Do global matrix multiply via small stiffness matrices,
c  gb=Au + b.  The long statement below correctly brings in all the
c  terms from the global matrix A using only the small stiffness matrices.

      do 3000 m=1,ns
      gb(m)=gb(m)+u(ib(m,1))*( dk(pix(ib(m,27)),1,4)+
     &dk(pix(ib(m,7)),2,3)+
     &dk(pix(ib(m,25)),5,8)+dk(pix(ib(m,15)),6,7) )+
     &u(ib(m,2))*( dk(pix(ib(m,27)),1,3)+dk(pix(ib(m,25)),5,7) )+
     &u(ib(m,3))*( dk(pix(ib(m,27)),1,2)+dk(pix(ib(m,5)),4,3)+
     &dk(pix(ib(m,13)),8,7)+dk(pix(ib(m,25)),5,6) )+
     &u(ib(m,4))*( dk(pix(ib(m,5)),4,2)+dk(pix(ib(m,13)),8,6) )+
     &u(ib(m,5))*( dk(pix(ib(m,6)),3,2)+dk(pix(ib(m,5)),4,1)+
     &dk(pix(ib(m,14)),6,7)+dk(pix(ib(m,13)),8,5) )+
     &u(ib(m,6))*( dk(pix(ib(m,6)),3,1)+dk(pix(ib(m,14)),7,5) )+
     &u(ib(m,7))*( dk(pix(ib(m,6)),3,4)+dk(pix(ib(m,7)),2,1)+
     &dk(pix(ib(m,14)),7,8)+dk(pix(ib(m,15)),6,5) )+
     &u(ib(m,8))*( dk(pix(ib(m,7)),2,4)+dk(pix(ib(m,15)),6,8) )+
     &u(ib(m,9))*( dk(pix(ib(m,25)),5,4)+dk(pix(ib(m,15)),6,3) )+
     &u(ib(m,10))*( dk(pix(ib(m,25)),5,3) )+
     &u(ib(m,11))*( dk(pix(ib(m,13)),8,3)+dk(pix(ib(m,25)),5,2) )+
     &u(ib(m,12))*( dk(pix(ib(m,13)),8,2) )+
     &u(ib(m,13))*( dk(pix(ib(m,13)),8,1)+dk(pix(ib(m,14)),7,2) )+
     &u(ib(m,14))*( dk(pix(ib(m,14)),7,1) )+
     &u(ib(m,15))*( dk(pix(ib(m,14)),7,4)+dk(pix(ib(m,15)),6,1) )+
     &u(ib(m,16))*( dk(pix(ib(m,15)),6,4) )+
     &u(ib(m,17))*( dk(pix(ib(m,27)),1,8)+dk(pix(ib(m,7)),2,7) )+
     &u(ib(m,18))*( dk(pix(ib(m,27)),1,7) )+
     &u(ib(m,19))*( dk(pix(ib(m,27)),1,6)+dk(pix(ib(m,5)),4,7) )+
     &u(ib(m,20))*( dk(pix(ib(m,5)),4,6) )+
     &u(ib(m,21))*( dk(pix(ib(m,5)),4,5)+dk(pix(ib(m,6)),3,6) )+
     &u(ib(m,22))*( dk(pix(ib(m,6)),3,5) )+
     &u(ib(m,23))*( dk(pix(ib(m,6)),3,8)+dk(pix(ib(m,7)),2,5) )+
     &u(ib(m,24))*( dk(pix(ib(m,7)),2,8) )+
     &u(ib(m,25))*( dk(pix(ib(m,14)),7,3)+dk(pix(ib(m,13)),8,4)+
     &dk(pix(ib(m,15)),6,2)+dk(pix(ib(m,25)),5,1) )+
     &u(ib(m,26))*( dk(pix(ib(m,6)),3,7)+dk(pix(ib(m,5)),4,8)+
     &dk(pix(ib(m,27)),1,5)+dk(pix(ib(m,7)),2,6) )+
     &u(ib(m,27))*( dk(pix(ib(m,27)),1,1)+dk(pix(ib(m,7)),2,2)+
     &dk(pix(ib(m,6)),3,3)+dk(pix(ib(m,5)),4,4)+dk(pix(ib(m,25)),5,5)+
     &dk(pix(ib(m,15)),6,6)+dk(pix(ib(m,14)),7,7)+
     &dk(pix(ib(m,13)),8,8) )
3000  continue

utot=0.0
do 3100 m=1,ns
utot=utot+0.5*u(m)*gb(m)+b(m)*u(m)
gb(m)=gb(m)+b(m)
3100 continue

utot=utot+C

        return
        end

c     Subroutine that carries out the conjugate gradient relaxation process

      subroutine dembx(ns,Lstep,gg,dk,gtest,ldemb,kkk)
      real u(8000),gb(8000),dk(100,8,8)
      real Ah(8000),h(8000),B,lambda,gamma
      integer*4 ib(8000,27)
      integer*2 pix(8000)

common/list3/ib
        common/list4/pix
common/list6/u
common/list7/gb
common/list9/h,Ah
```

```
c  Initialize the conjugate direction vector on first call to dembx only.
c  For calls to dembx after the first, we want to continue using the
c  value fo h determined in the previous call.  Of course, if npooints
c  is greater than 1, then this initialization step will be run every
c  a new microstructure is used, as kkk will be reset to 1 every time
c  the counter micro is increased.
      if(kkk.eq.1) then
      do 50 m=1,ns
      h(m)=gb(m)
50    continue
      end if
c  Lstep counts the number of conjugate gradient steps taken in each call
c  to dembx.
      Lstep=0

c     Conjugate gradient loop

      do 800 ijk=1,ldemb
      Lstep=Lstep+1

      do 290 m=1,ns
      Ah(m)=0.0
290   continue

c  Do global matrix multiply via small stiffness matrices, Ah = A * h.
c  The long statement below correctly brings in all the terms from
c  the global matrix A using only the small stiffness matrices.

      do 400 m=1,ns
      Ah(m)=Ah(m)+h(ib(m,1))*( dk(pix(ib(m,27)),1,4)+
     &dk(pix(ib(m,7)),2,3)+
     &dk(pix(ib(m,25)),5,8)+dk(pix(ib(m,15)),6,7) )+
     &h(ib(m,2))*( dk(pix(ib(m,27)),1,3)+dk(pix(ib(m,25)),5,7) )+
     &h(ib(m,3))*( dk(pix(ib(m,27)),1,2)+dk(pix(ib(m,5)),4,3)+
     &dk(pix(ib(m,13)),8,7)+dk(pix(ib(m,25)),5,6) )+
     &h(ib(m,4))*( dk(pix(ib(m,5)),4,2)+dk(pix(ib(m,13)),8,6) )+
     &h(ib(m,5))*( dk(pix(ib(m,6)),3,2)+dk(pix(ib(m,5)),4,1)+
     &dk(pix(ib(m,14)),6,7)+dk(pix(ib(m,13)),8,5) )+
     &h(ib(m,6))*( dk(pix(ib(m,6)),3,1)+dk(pix(ib(m,14)),7,5) )+
     &h(ib(m,7))*( dk(pix(ib(m,6)),3,4)+dk(pix(ib(m,7)),2,1)+
     &dk(pix(ib(m,14)),7,8)+dk(pix(ib(m,15)),6,5) )+
     &h(ib(m,8))*( dk(pix(ib(m,7)),2,4)+dk(pix(ib(m,15)),6,8) )+
     &h(ib(m,9))*( dk(pix(ib(m,25)),5,4)+dk(pix(ib(m,15)),6,3) )+
     &h(ib(m,10))*( dk(pix(ib(m,25)),5,3) )+
    &h(ib(m,11))*( dk(pix(ib(m,13)),8,3)+dk(pix(ib(m,25)),5,2) )+
    &h(ib(m,12))*( dk(pix(ib(m,13)),8,2) )+
    &h(ib(m,13))*( dk(pix(ib(m,13)),8,1)+dk(pix(ib(m,14)),7,2) )+
    &h(ib(m,14))*( dk(pix(ib(m,14)),7,1) )+
    &h(ib(m,15))*( dk(pix(ib(m,14)),7,4)+dk(pix(ib(m,15)),6,1) )+
    &h(ib(m,16))*( dk(pix(ib(m,15)),6,4) )+
    &h(ib(m,17))*( dk(pix(ib(m,27)),1,8)+dk(pix(ib(m,7)),2,7) )+
    &h(ib(m,18))*( dk(pix(ib(m,27)),1,7) )+
    &h(ib(m,19))*( dk(pix(ib(m,27)),1,6)+dk(pix(ib(m,5)),4,7) )+
    &h(ib(m,20))*( dk(pix(ib(m,5)),4,6) )+
    &h(ib(m,21))*( dk(pix(ib(m,5)),4,5)+dk(pix(ib(m,6)),3,6) )+
    &h(ib(m,22))*( dk(pix(ib(m,6)),3,5) )+
    &h(ib(m,23))*( dk(pix(ib(m,6)),3,8)+dk(pix(ib(m,7)),2,5) )+
    &h(ib(m,24))*( dk(pix(ib(m,7)),2,8) )+
    &h(ib(m,25))*( dk(pix(ib(m,14)),7,3)+dk(pix(ib(m,13)),8,4)+
    &dk(pix(ib(m,15)),6,2)+dk(pix(ib(m,25)),5,1) )+
    &h(ib(m,26))*( dk(pix(ib(m,6)),3,7)+dk(pix(ib(m,5)),4,8)+
    &dk(pix(ib(m,27)),1,5)+dk(pix(ib(m,7)),2,6) )+
    &h(ib(m,27))*( dk(pix(ib(m,27)),1,1)+dk(pix(ib(m,7)),2,2)+
    &dk(pix(ib(m,6)),3,3)+dk(pix(ib(m,5)),4,4)+dk(pix(ib(m,25)),5,5)+
    &dk(pix(ib(m,15)),6,6)+dk(pix(ib(m,14)),7,7)+
    &dk(pix(ib(m,13)),8,8) )

400   continue

      hAh=0.0
      do 530 m=1,ns
      hAh=hAh+h(m)*Ah(m)
530   continue

      lambda=gg/hAh
      do 540 m=1,ns
      u(m)=u(m)-lambda*h(m)
      gb(m)=gb(m)-lambda*Ah(m)
540   continue

      gglast=gg
      gg=0.0
      do 550 m=1,ns
      gg=gg+gb(m)*gb(m)
550   continue
      if(gg.le.gtest) goto 1000

      gamma=gg/gglast
      do 570 m=1,ns
      h(m)=gb(m)+gamma*h(m)
570   continue

800   continue

1000  continue

      return
      end

c Subroutine that computes average current in three directions

      subroutine current(nx,ny,nz,ns)

      real af(3,8)
      real u(8000),uu(8)
      real sigma(100,3,3)
      integer*4 ib(8000,27)
      integer*2 pix(8000)

common/list1/currx,curry,currz
common/list2/ex,ey,ez
common/list3/ib
        common/list4/pix
common/list6/u
common/list8/sigma

      nxy=nx*ny
c  af is the average field matrix, average field in a pixel is af*u(pixel).
c  The matrix af relates the nodal voltages to the average field in the pixel.

c Set up single element average field matrix

      af(1,1)=0.25
      af(1,2)=-0.25
      af(1,3)=-0.25
      af(1,4)=0.25
      af(1,5)=0.25
      af(1,6)=-0.25
      af(1,7)=-0.25
      af(1,8)=0.25
      af(2,1)=0.25
      af(2,2)=0.25
      af(2,3)=-0.25
      af(2,4)=-0.25
      af(2,5)=0.25
      af(2,6)=0.25
      af(2,7)=-0.25
      af(2,8)=-0.25
      af(3,1)=0.25
      af(3,2)=0.25
      af(3,3)=0.25
      af(3,4)=0.25
      af(3,5)=-0.25
      af(3,6)=-0.25
      af(3,7)=-0.25
      af(3,8)=-0.25

c  now compute current in each pixel
      currx=0.0
      curry=0.0
      currz=0.0
c  compute average field in each pixel
      do 470 k=1,nz
      do 470 j=1,ny
      do 470 i=1,nx
      m=(k-1)*nxy+(j-1)*nx+i
c  load in elements of 8-vector using pd. bd. conds.
      uu(1)=u(m)
      uu(2)=u(ib(m,3))
      uu(3)=u(ib(m,2))
      uu(4)=u(ib(m,1))
      uu(5)=u(ib(m,26))
      uu(6)=u(ib(m,19))
      uu(7)=u(ib(m,18))
      uu(8)=u(ib(m,17))
c  Correct for periodic boundary conditions, some voltages are wrong
c  for a pixel on a periodic boundary. Since they come from an opposite
c  face, need to put in applied fields to correct them.
      if(i.eq.nx) then
      uu(2)=uu(2)-ex*nx
      uu(3)=uu(3)-ex*nx
      uu(6)=uu(6)-ex*nx
      uu(7)=uu(7)-ex*nx
      end if
      if(j.eq.ny) then
      uu(3)=uu(3)-ey*ny
      uu(4)=uu(4)-ey*ny
      uu(7)=uu(7)-ey*ny
      uu(8)=uu(8)-ey*ny
      end if
      if(k.eq.nz) then
      uu(5)=uu(5)-ez*nz
      uu(6)=uu(6)-ez*nz
      uu(7)=uu(7)-ez*nz
      uu(8)=uu(8)-ez*nz
      end if
c  cur1, cur2, cur3 are the local currents averaged over the pixel
      cur1=0.0
      cur2=0.0
      cur3=0.0
      do 465 n=1,8
      do 465 nn=1,3
      cur1=cur1+sigma(pix(m),1,nn)*af(nn,n)*uu(n)
      cur2=cur2+sigma(pix(m),2,nn)*af(nn,n)*uu(n)
      cur3=cur3+sigma(pix(m),3,nn)*af(nn,n)*uu(n)
465   continue
c  sum into the global average currents
      currx=currx+cur1
      curry=curry+cur2
      currz=currz+cur3
470   continue

c Volume average currents
      currx=currx/float(ns)
      curry=curry/float(ns)
      currz=currz/float(ns)

      return
      end

c  Subroutine that counts phase volume fractions

      subroutine assig(ns,nphase,prob)

      integer ns,nphase
      integer*2 pix(8000)
      real prob(100)
      common/list4/pix

do 90 i=1,nphase
prob(i)=0.0
90      continue

      do 100 m=1,ns
do 100 i=1,nphase
        if(pix(m).eq.i) then
prob(i)=prob(i)+1
endif
100   continue

do 110 i=1,nphase
prob(i)=prob(i)/float(ns)
110     continue

      return
      end

c  Subroutine that sets up microstructural image

      subroutine ppixel(nx,ny,nz,ns,nphase)
      integer*2 pix(8000)
      common/list4/pix

c  (USER) If you want to set up a test image inside the program, instead
c  of reading it in from a file, this should be done inside this subroutine.

      do 100 k=1,nz
      do 100 j=1,ny
      do 100 i=1,nx
      m=nx*ny*(k-1)+nx*(j-1)+i
      read(9,*) pix(m)
100   continue

c  Check for wrong phase labels--less than 1 or greater than nphase
       do 500 m=1,ns
       if(pix(m).lt.1) then
        write(7,*) 'Phase label in pix < 1--error at ',m
       end if
       if(pix(m).gt.nphase) then
        write(7,*) 'Phase label in pix > nphase--error at ',m
       end if
500    continue

      return
      end
```

9.3.2 ELAS3D.F

```
c  ***********************  elas3d.f  ***********************************
c  BACKGROUND

c  This program solves the linear elastic equations in a
c  random linear elastic material, subject to an applied macroscopic strain,
c  using the finite element method.  Each pixel in the 3-D digital
c  image is a cubic tri-linear finite element,  having its own
c  elastic moduli tensor. Periodic boundary conditions are maintained.
c  In the comments below, (USER) means that this is a section of code that
c  the user might have to change for his particular problem. Therefore the
c  user is encouraged to search for this string.

c  PROBLEM AND VARIABLE DEFINITION

c  The problem being solved is the minimization of the energy
c  1/2 uAu + b u + C, where A is the Hessian matrix composed of the
c  stiffness matrices (dk) for each pixel/element, b is a constant vector
c  and C is a constant that are determined by the applied strain and
c  the periodic boundary conditions, and u is a vector of
c  all the displacements. The solution
c  method used is the conjugate gradient relaxation algorithm.
c  Other variables are:  gb is the gradient = Au+b, h and Ah are
c  auxiliary variables used in the conjugate gradient algorithm (in dembx),
c  dk(n,i,j) is the stiffness matrix of the n'th phase, cmod(n,i,j) is
c  the elastic moduli tensor of the n'th phase, pix is a vector that gives
c  the phase label of each pixel, ib is a matrix that gives the labels of
c  the 27 (counting itself) neighbors of a given node, prob is the volume
c  fractions of the various phases,
c  strxx, stryy, strzz, strxz, stryz, and strxy are the six Voigt
c  volume averaged total stresses, and
c  sxx, syy, szz, sxz, syz, and sxy are the six Voigt
c  volume averaged total strains.

c  DIMENSIONS

c  The vectors u,gb,b,h, and Ah are dimensioned to be the system size,
c  ns=nx*ny*nz, with three components, where the digital image of the
c  microstructure considered is a rectangular paralleliped, nx x ny x nz
c  in size.  The arrays pix and ib are are also dimensioned to the system size.
c  The array ib has 27 components, for the 27 neighbors of a node.
c  Note that the program is set up at present to have at most 100
c  different phases.  This can easily be changed, simply by changing
c  the dimensions of dk, prob, and cmod. The parameter nphase gives the
```

```
c  number of phases being considered in the problem.
c  All arrays are passed between subroutines using simple common statements.

c  STRONGLY SUGGESTED:  READ THE MANUAL BEFORE USING PROGRAM!!

c  (USER) Change these dimensions and in other subroutines at same time.
c  For example, search and replace all occurrences throughout the
c  program of "(8000" by "(64000", to go from a
c  20 x 20 x 20 system to a 40 x 40 x 40 system.
real u(8000,3),gb(8000,3),b(8000,3)
        real h(8000,3),Ah(8000,3)
real cmod(100,6,6),dk(100,8,3,8,3)
real phasemod(100,2),prob(100)
integer in(27),jn(27),kn(27)
integer*4 ib(8000,27)
integer*2 pix(8000)

common/list1/strxx,stryy,strzz,strxz,stryz,strxy
common/list2/exx,eyy,ezz,exz,eyz,exy
common/list3/ib
        common/list4/pix
common/list5/dk,b,C
common/list6/u
common/list7/gb
        common/list8/cmod
        common/list9/h,Ah
common/list10/sxx,syy,szz,sxz,syz,sxy

c (USER) Unit 9 is the microstructure input file,
c  unit 7 is the results output file.
        open (unit=9,file='microstructure.dat')
        open (unit=7,file='outputfile.out')

c (USER)  nx,ny,nz give the size of the lattice
        nx=20
        ny=20
        nz=20
c ns=total number of sites
        ns=nx*ny*nz
      write(7,9010) nx,ny,nz,ns
9010  format('nx= ',i4,' ny= ',i4,' nz= ',i4,' ns= 'i8)

c  (USER) nphase is the number of phases being considered in the problem.
c  The values of pix(m) will run from 1 to nphase.
nphase=2
```

```
c  (USER) gtest is the stopping criterion, the number
c  to which the quantity gg=gb*gb is compared.
c  Usually gtest = abc*ns, so that when gg < gtest, the rms value
c  per pixel of gb is less than sqrt(abc).
        gtest=1.e-12*ns

c  (USER)
c  The parameter phasemod(i,j) is the bulk (i,1) and shear (i,2) moduli of
c  the i'th phase. These can be input in terms of Young's moduli E(i,1) and
c  Poisson's ratio nu (i,2).  The program, in do loop 1144, then changes them
c  to bulk and shear moduli, using relations for isotropic elastic
c  moduli.  For anisotropic elastic material, one can directly input
c  the elastic moduli tensor cmod in subroutine femat, and skip this part.
c  If you wish to input in terms of bulk (1) and shear (2), then make sure
c  to comment out the do 1144 loop.
phasemod(1,1)=1.0
phasemod(1,2)=0.2
phasemod(2,1)=0.5
phasemod(2,2)=0.2

c  (USER) Program uses bulk modulus (1) and shear modulus (2), so transform
c  Young's modulis (1) and Poisson's ratio (2).
        do 1144 i=1,nphase
        save=phasemod(i,1)
        phasemod(i,1)=phasemod(i,1)/3./(1.-2.*phasemod(i,2))
        phasemod(i,2)=save/2./(1.+phasemod(i,2))
1144    continue

c  Construct the neighbor table, ib(m,n)

c  First construct the 27 neighbor table in terms of delta i, delta j, and
c  delta k information (see Table 3 in manual)
      in(1)=0
      in(2)=1
      in(3)=1
      in(4)=1
      in(5)=0
      in(6)=-1
      in(7)=-1
      in(8)=-1

      jn(1)=1
      jn(2)=1
      jn(3)=0
      jn(4)=-1
      jn(5)=-1
      jn(6)=-1
      jn(7)=0
      jn(8)=1

      do 555 n=1,8
      kn(n)=0
      kn(n+8)=-1
      kn(n+16)=1
      in(n+8)=in(n)
      in(n+16)=in(n)
      jn(n+8)=jn(n)
      jn(n+16)=jn(n)
555   continue
      in(25)=0
      in(26)=0
      in(27)=0
      jn(25)=0
      jn(26)=0
      jn(27)=0
      kn(25)=-1
      kn(26)=1
      kn(27)=0

c  Now construct neighbor table according to 1-d labels
c  Matrix ib(m,n) gives the 1-d label of the n'th neighbor (n=1,27) of
c  the node labelled m.
      nxy=nx*ny
      do 1020 k=1,nz
      do 1020 j=1,ny
      do 1020 i=1,nx
      m=nxy*(k-1)+nx*(j-1)+i
      do 1004 n=1,27
      i1=i+in(n)
      j1=j+jn(n)
      k1=k+kn(n)
      if(i1.lt.1) i1=i1+nx
      if(i1.gt.nx) i1=i1-nx
      if(j1.lt.1) j1=j1+ny
      if(j1.gt.ny) j1=j1-ny
      if(k1.lt.1) k1=k1+nz
      if(k1.gt.nz) k1=k1-nz
      m1=nxy*(k1-1)+nx*(j1-1)+i1
      ib(m,n)=m1
```

```
1004  continue
1020  continue

c Compute the average stress and strain in each microstructure.
c (USER) npoints is the number of microstructures to use.

        npoints=1
        do 8000 micro=1,npoints
c  Read in a microstructure in subroutine ppixel, and set up pix(m)
c  with the appropriate phase assignments.
        call ppixel(nx,ny,nz,ns,nphase)
c Count and output the volume fractions of the different phases
        call assig(ns,nphase,prob)
        do 111 i=1,nphase
        write(7,9020) i,phasemod(i,1),phasemod(i,2)
9020    format(' Phase ',i3,' bulk = ',f12.6,' shear = ',f12.6)
111 continue

do 8050 i=1,nphase
write(7,9065) i,prob(i)
9065 format(' Volume fraction of phase ',i3,'  is ',f8.5)
8050 continue

c  (USER) Set applied strains
c  Actual shear strain applied in do 1050 loop is exy, exz, and eyz as
c  given in the statements below.  The engineering shear strain, by which
c  the shear modulus is usually defined, is twice these values.
        exx=0.1
        eyy=0.1
        ezz=0.1
        exz=0.1/2.
        eyz=0.2/2.
        exy=0.3/2.
        write(7,*) 'Applied engineering strains'
        write(7,*) ' exx eyy ezz exz eyz exy'
        write(7,*) exx,eyy,ezz,2.*exz,2.*eyz,2.*exy

c Set up the elastic modulus variables, finite element stiffness matrices,
c the constant, C, and vector, b, required for computing the energy.
c  (USER) If anisotropic elastic moduli tensors are used, these need to be
c  input in subroutine femat.

call femat(nx,ny,nz,ns,phasemod,nphase)

c Apply chosen strains as a homogeneous macroscopic strain
```

```
c as the initial condition.
do 1050 k=1,nz
do 1050 j=1,ny
        do 1050 i=1,nx
m=nxy*(k-1)+nx*(j-1)+i
x=float(i-1)
y=float(j-1)
z=float(k-1)
u(m,1)=x*exx+y*exy+z*exz
                u(m,2)=x*exy+y*eyy+z*eyz
                u(m,3)=x*exz+y*eyz+z*ezz
1050 continue

c  RELAXATION LOOP
c  (USER) kmax is the maximum number of times dembx will be called, with
c  ldemb conjugate gradient steps performed during each call.  The total
c  number of conjugate gradient steps allowed for a given elastic
c  computation is kmax*ldemb.
        kmax=40
        ldemb=50
        ltot=0
c  Call energy to get initial energy and initial gradient
        call energy(nx,ny,nz,ns,utot)
c  gg is the norm squared of the gradient (gg=gb*gb)
  gg=0.0
        do 100 m3=1,3
        do 100 m=1,ns
        gg=gg+gb(m,m3)*gb(m,m3)
100     continue
write(7,*) 'Initial energy = ',utot,' gg = ',gg
        call flush(7)

        do 5000 kkk=1,kmax
c  call dembx to go into the conjugate gradient solver
        call dembx(ns,Lstep,gg,dk,gtest,ldemb,kkk)
        ltot=ltot+Lstep
c  Call energy to compute energy after dembx call. If gg < gtest, this
c  will be the final energy.  If gg is still larger than gtest, then this
c  will give an intermediate energy with which to check how the
c  relaxation process is coming along.
        call energy(nx,ny,nz,ns,utot)
write(7,*) 'Energy = ',utot,' gg = ',gg
write(7,*) 'Number of conjugate steps  = ',ltot
        call flush(7)
c  If relaxation process is finished, jump out of loop
```

```
        if(gg.le.gtest) goto 444
c  If relaxation process will continue, compute and output stresses
c  and strains as an additional aid to judge how the
c  relaxation procedure is progressing.
call stress(nx,ny,nz,ns)
        write(7,*) ' stresses:  xx,yy,zz,xz,yz,xy'
write(7,*) strxx,stryy,strzz,strxz,stryz,strxy
        write(7,*) ' strains:  xx,yy,zz,xz,yz,xy'
write(7,*) sxx,syy,szz,sxz,syz,sxy
5000    continue

444     call stress(nx,ny,nz,ns)
        write(7,*) ' stresses:  xx,yy,zz,xz,yz,xy'
write(7,*) strxx,stryy,strzz,strxz,stryz,strxy
        write(7,*) ' strains:  xx,yy,zz,xz,yz,xy'
write(7,*) sxx,syy,szz,sxz,syz,sxy

8000    continue

        end

c  Subroutine that sets up the elastic moduli variables,
c  the stiffness matrices,dk, the linear term in
c  displacements, b, and the constant term, C, that appear in the total energy
c  due to the periodic boundary conditions

      subroutine femat(nx,ny,nz,ns,phasemod,nphase)
      real dk(100,8,3,8,3),phasemod(100,2),dndx(8),dndy(8),dndz(8)
      real b(8000,3),g(3,3,3),C,ck(6,6),cmu(6,6),cmod(100,6,6)
      real es(6,8,3),delta(8,3)
      integer is(8)
      integer*4 ib(8000,27)
      integer*2 pix(8000)

common/list2/exx,eyy,ezz,exz,eyz,exy
common/list3/ib
        common/list4/pix
common/list5/dk,b,C
        common/list8/cmod

      nxy=nx*ny

c  (USER) NOTE:  complete elastic modulus matrix is used, so an anisotropic
c  matrix could be directly input at any point, since program is written
c  to use a general elastic moduli tensor, but is only explicitly
```

```
c  implemented for isotropic materials.

c  initialize stiffness matrices
      do 40 m=1,nphase
      do 40 l=1,3
      do 40 k=1,3
      do 40 j=1,8
      do 40 i=1,8
      dk(m,i,k,j,l)=0.0
40    continue

c set up elastic moduli matrices for each kind of element
c  ck and cmu are the bulk and shear modulus matrices, which need to be
c  weighted by the actual bulk and shear moduli

      ck(1,1)=1.0
      ck(1,2)=1.0
      ck(1,3)=1.0
      ck(1,4)=0.0
      ck(1,5)=0.0
      ck(1,6)=0.0
      ck(2,1)=1.0
      ck(2,2)=1.0
      ck(2,3)=1.0
      ck(2,4)=0.0
      ck(2,5)=0.0
      ck(2,6)=0.0
      ck(3,1)=1.0
      ck(3,2)=1.0
      ck(3,3)=1.0
      ck(3,4)=0.0
      ck(3,5)=0.0
      ck(3,6)=0.0
      ck(4,1)=0.0
      ck(4,2)=0.0
      ck(4,3)=0.0
      ck(4,4)=0.0
      ck(4,5)=0.0
      ck(4,6)=0.0
      ck(5,1)=0.0
      ck(5,2)=0.0
      ck(5,3)=0.0
      ck(5,4)=0.0
      ck(5,5)=0.0
      ck(5,6)=0.0
ck(6,1)=0.0
ck(6,2)=0.0
ck(6,3)=0.0
ck(6,4)=0.0
ck(6,5)=0.0
ck(6,6)=0.0

cmu(1,1)=4.0/3.0
cmu(1,2)=-2.0/3.0
cmu(1,3)=-2.0/3.0
cmu(1,4)=0.0
cmu(1,5)=0.0
cmu(1,6)=0.0
cmu(2,1)=-2.0/3.0
cmu(2,2)=4.0/3.0
cmu(2,3)=-2.0/3.0
cmu(2,4)=0.0
cmu(2,5)=0.0
cmu(2,6)=0.0
cmu(3,1)=-2.0/3.0
cmu(3,2)=-2.0/3.0
cmu(3,3)=4.0/3.0
cmu(3,4)=0.0
cmu(3,5)=0.0
cmu(3,6)=0.0
cmu(4,1)=0.0
cmu(4,2)=0.0
cmu(4,3)=0.0
cmu(4,4)=1.0
cmu(4,5)=0.0
cmu(4,6)=0.0
cmu(5,1)=0.0
cmu(5,2)=0.0
cmu(5,3)=0.0
cmu(5,4)=0.0
cmu(5,5)=1.0
cmu(5,6)=0.0
cmu(6,1)=0.0
cmu(6,2)=0.0
cmu(6,3)=0.0
cmu(6,4)=0.0
cmu(6,5)=0.0
cmu(6,6)=1.0

do 31 k=1,nphase
```

```
      do 21 j=1,6
      do 11 i=1,6
      cmod(k,i,j)=phasemod(k,1)*ck(i,j)+phasemod(k,2)*cmu(i,j)
11    continue
21    continue
31    continue

c  set up Simpson's integration rule weight vector
      do 30 k=1,3
      do 30 j=1,3
      do 30 i=1,3
      nm=0
      if(i.eq.2) nm=nm+1
      if(j.eq.2) nm=nm+1
      if(k.eq.2) nm=nm+1
      g(i,j,k)=4.0**nm
30    continue

c  loop over the nphase kinds of pixels and Simpson's rule quadrature
c  points in order to compute the stiffness matrices.  Stiffness matrices
c  of trilinear finite elements are quadratic in x, y, and z, so that
c  Simpson's rule quadrature gives exact results.
      do 4000 ijk=1,nphase
      do 3000 k=1,3
      do 3000 j=1,3
      do 3000 i=1,3
      x=float(i-1)/2.0
      y=float(j-1)/2.0
      z=float(k-1)/2.0
c  dndx means the negative derivative, with respect to x, of the shape
c  matrix N (see manual, Sec. 2.2), dndy, and dndz are similar.
      dndx(1)=-(1.0-y)*(1.0-z)
      dndx(2)=(1.0-y)*(1.0-z)
      dndx(3)=y*(1.0-z)
      dndx(4)=-y*(1.0-z)
      dndx(5)=-(1.0-y)*z
      dndx(6)=(1.0-y)*z
      dndx(7)=y*z
      dndx(8)=-y*z
      dndy(1)=-(1.0-x)*(1.0-z)
      dndy(2)=-x*(1.0-z)
      dndy(3)=x*(1.0-z)
      dndy(4)=(1.0-x)*(1.0-z)
      dndy(5)=-(1.0-x)*z
      dndy(6)=-x*z
      dndy(7)=x*z
      dndy(8)=(1.0-x)*z
      dndz(1)=-(1.0-x)*(1.0-y)
      dndz(2)=-x*(1.0-y)
      dndz(3)=-x*y
      dndz(4)=-(1.0-x)*y
      dndz(5)=(1.0-x)*(1.0-y)
      dndz(6)=x*(1.0-y)
      dndz(7)=x*y
      dndz(8)=(1.0-x)*y
c  now build strain matrix
      do 2799 n1=1,6
      do 2799 n2=1,8
      do 2799 n3=1,3
      es(n1,n2,n3)=0.0
2799  continue
      do 2797 n=1,8
      es(1,n,1)=dndx(n)
      es(2,n,2)=dndy(n)
      es(3,n,3)=dndz(n)
      es(4,n,1)=dndz(n)
      es(4,n,3)=dndx(n)
      es(5,n,2)=dndz(n)
      es(5,n,3)=dndy(n)
      es(6,n,1)=dndy(n)
      es(6,n,2)=dndx(n)
2797  continue
c  Matrix multiply to determine value at (x,y,z), multiply by
c  proper weight, and sum into dk, the stiffness matrix
      do 900 mm=1,3
      do 900 nn=1,3
      do 900 ii=1,8
      do 900 jj=1,8
c  Define sum over strain matrices and elastic moduli matrix for
c  stiffness matrix
      sum=0.0
      do 890 kk=1,6
      do 890 ll=1,6
      sum=sum+es(kk,ii,mm)*cmod(ijk,kk,ll)*es(ll,jj,nn)
890   continue
      dk(ijk,ii,mm,jj,nn)=dk(ijk,ii,mm,jj,nn)+g(i,j,k)*sum/216.
900   continue
3000  continue
4000  continue
```

```
c  Set up vector for linear term, b, and constant term, C,
c  in the elastic energy.  This is done using the stiffness matrices,
c  and the periodic terms in the applied strain that come in at the
c  boundary pixels via the periodic boundary conditions and the
c  condition that an applied macroscopic strain exists (see Sec. 2.2
c  in the manual). It is easier to set b up this way than to analytically
c  write out all the terms involved.

c  Initialize b and C
      do 5000 m3=1,3
      do 5000 m=1,ns
      b(m,m3)=0.0
5000  continue
      C=0.0

c  For all cases, the correspondence between 1-8 finite element node
c  labels and 1-27 neighbor labels is (see Table 4 in manual):
c  1:ib(m,27), 2:ib(m,3),
c  3:ib(m,2),4:ib(m,1),
c  5:ib(m,26),6:ib(m,19)
c  7:ib(m,18),8:ib(m,17)
      is(1)=27
      is(2)=3
      is(3)=2
      is(4)=1
      is(5)=26
      is(6)=19
      is(7)=18
      is(8)=17

c  x=nx face
      do 2001 i3=1,3
      do 2001 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
2001  continue
      do 2000 j=1,ny-1
      do 2000 k=1,nz-1
      m=nxy*(k-1)+j*nx
      do 1900 nn=1,3
      do 1900 mm=1,8
      sum=0.0
      do 1899 m3=1,3
      do 1899 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
      C=C+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
1899  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1900  continue
2000  continue
c  y=ny face
      do 2011 i3=1,3
      do 2011 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
2011  continue
      do 2010 i=1,nx-1
      do 2010 k=1,nz-1
      m=nxy*(k-1)+nx*(ny-1)+i
      do 1901 nn=1,3
      do 1901 mm=1,8
      sum=0.0
      do 2099 m3=1,3
      do 2099 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
      C=C+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2099  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1901  continue
2010  continue
c  z=nz face
      do 2021 i3=1,3
      do 2021 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
2021  continue
      do 2020 i=1,nx-1
      do 2020 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      do 1902 nn=1,3
      do 1902 mm=1,8
      sum=0.0
      do 2019 m3=1,3
      do 2019 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
      C=C+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2019  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1902  continue
2020  continue
c  x=nx y=ny edge
      do 2031 i3=1,3
      do 2031 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.6) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      delta(i8,1)=exy*ny+exx*nx
      delta(i8,2)=eyy*ny+exy*nx
      delta(i8,3)=eyz*ny+exz*nx
      end if
2031  continue
      do 2030 k=1,nz-1
      m=nxy*k
      do 1903 nn=1,3
      do 1903 mm=1,8
      sum=0.0
      do 2029 m3=1,3
      do 2029 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
      C=C+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2029  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1903  continue
2030  continue
```

```
c  x=nx z=nz edge
      do 2041 i3=1,3
      do 2041 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.5.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exz*nz+exx*nx
      delta(i8,2)=eyz*nz+exy*nx
      delta(i8,3)=ezz*nz+exz*nx
      end if
2041  continue
      do 2040 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      do 1904 nn=1,3
      do 1904 mm=1,8
      sum=0.0
      do 2039 m3=1,3
      do 2039 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
      C=C+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2039  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1904  continue
2040  continue
c  y=ny z=nz edge
      do 2051 i3=1,3
      do 2051 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.3.or.i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
2051  continue
      do 2050 i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      do 1905 nn=1,3
      do 1905 mm=1,8
      sum=0.0
      do 2049 m3=1,3
      do 2049 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
      C=C+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2049  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1905  continue
2050  continue
c  x=nx y=ny z=nz corner
      do 2061 i3=1,3
      do 2061 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.5) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      if(i8.eq.6) then
      delta(i8,1)=exx*nx+exz*nz
      delta(i8,2)=exy*nx+eyz*nz
      delta(i8,3)=exz*nx+ezz*nz
      end if
      if(i8.eq.3) then
      delta(i8,1)=exx*nx+exy*ny
      delta(i8,2)=exy*nx+eyy*ny
      delta(i8,3)=exz*nx+eyz*ny
      end if
      if(i8.eq.7) then
      delta(i8,1)=exx*nx+exy*ny+exz*nz
      delta(i8,2)=exy*nx+eyy*ny+eyz*nz
      delta(i8,3)=exz*nx+eyz*ny+ezz*nz
      end if
2061  continue
      m=nx*ny*nz
      do 1906 nn=1,3
      do 1906 mm=1,8
      sum=0.0
      do 2059 m3=1,3
      do 2059 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
      C=C+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2059  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1906  continue

      return
      end

c  Subroutine computes the total energy, utot, and the gradient, gb

      subroutine energy(nx,ny,nz,ns,utot)

real u(8000,3),gb(8000,3)
real b(8000,3),C,utot
real dk(100,8,3,8,3)
  integer*4 ib(8000,27)
  integer*2 pix(8000)

common/list2/exx,eyy,ezz,exz,eyz,exy
common/list3/ib
        common/list4/pix
common/list5/dk,b,C
```

```
common/list6/u
common/list7/gb

        do 2090 m3=1,3
do 2090 m=1,ns
gb(m,m3)=0.0
2090 continue

c  Do global matrix multiply via small stiffness matrices, gb = A * u
c  The long statement below correctly brings in all the terms from
c  the global matrix A using only the small stiffness matrices.

        do 3000 j=1,3
        do 3000 n=1,3
do 3000 m=1,ns
      gb(m,j)=gb(m,j)+u(ib(m,1),n)*( dk(pix(ib(m,27)),1,j,4,n)
     &+dk(pix(ib(m,7)),2,j,3,n)
     &+dk(pix(ib(m,25)),5,j,8,n)+dk(pix(ib(m,15)),6,j,7,n) )+
     &u(ib(m,2),n)*( dk(pix(ib(m,27)),1,j,3,n)
     &+dk(pix(ib(m,25)),5,j,7,n) )+
     &u(ib(m,3),n)*( dk(pix(ib(m,27)),1,j,2,n)+dk(pix(ib(m,5)),4,j,3,n)+
     &dk(pix(ib(m,13)),8,j,7,n)+dk(pix(ib(m,25)),5,j,6,n) )+
     &u(ib(m,4),n)*( dk(pix(ib(m,5)),4,j,2,n)
     &+dk(pix(ib(m,13)),8,j,6,n) )+
     &u(ib(m,5),n)*( dk(pix(ib(m,6)),3,j,2,n)+dk(pix(ib(m,5)),4,j,1,n)+
     &dk(pix(ib(m,14)),7,j,6,n)+dk(pix(ib(m,13)),8,j,5,n) )+
     &u(ib(m,6),n)*( dk(pix(ib(m,6)),3,j,1,n)
     &+dk(pix(ib(m,14)),7,j,5,n) )+
     &u(ib(m,7),n)*( dk(pix(ib(m,6)),3,j,4,n)+dk(pix(ib(m,7)),2,j,1,n)+
     &dk(pix(ib(m,14)),7,j,8,n)+dk(pix(ib(m,15)),6,j,5,n) )+
     &u(ib(m,8),n)*( dk(pix(ib(m,7)),2,j,4,n)
     &+dk(pix(ib(m,15)),6,j,8,n) )+
     &u(ib(m,9),n)*( dk(pix(ib(m,25)),5,j,4,n)
     &+dk(pix(ib(m,15)),6,j,3,n) )+
     &u(ib(m,10),n)*( dk(pix(ib(m,25)),5,j,3,n) )+
     &u(ib(m,11),n)*( dk(pix(ib(m,13)),8,j,3,n)
     &+dk(pix(ib(m,25)),5,j,2,n) )+
     &u(ib(m,12),n)*( dk(pix(ib(m,13)),8,j,2,n) )+
     &u(ib(m,13),n)*( dk(pix(ib(m,13)),8,j,1,n)
     &+dk(pix(ib(m,14)),7,j,2,n) )+
     &u(ib(m,14),n)*( dk(pix(ib(m,14)),7,j,1,n) )+
     &u(ib(m,15),n)*( dk(pix(ib(m,14)),7,j,4,n)
     &+dk(pix(ib(m,15)),6,j,1,n) )+
     &u(ib(m,16),n)*( dk(pix(ib(m,15)),6,j,4,n) )+
     &u(ib(m,17),n)*( dk(pix(ib(m,27)),1,j,8,n)
     &+dk(pix(ib(m,7)),2,j,7,n) )+
     &u(ib(m,18),n)*( dk(pix(ib(m,27)),1,j,7,n) )+
     &u(ib(m,19),n)*( dk(pix(ib(m,27)),1,j,6,n)
     &+dk(pix(ib(m,5)),4,j,7,n) )+
     &u(ib(m,20),n)*( dk(pix(ib(m,5)),4,j,6,n) )+
     &u(ib(m,21),n)*( dk(pix(ib(m,5)),4,j,5,n)
     &+dk(pix(ib(m,6)),3,j,6,n) )+
     &u(ib(m,22),n)*( dk(pix(ib(m,6)),3,j,5,n) )+
     &u(ib(m,23),n)*( dk(pix(ib(m,6)),3,j,8,n)
     &+dk(pix(ib(m,7)),2,j,5,n) )+
     &u(ib(m,24),n)*( dk(pix(ib(m,7)),2,j,8,n) )+
     &u(ib(m,25),n)*( dk(pix(ib(m,14)),7,j,3,n)
     &+dk(pix(ib(m,13)),8,j,4,n)+
     &dk(pix(ib(m,15)),6,j,2,n)+dk(pix(ib(m,25)),5,j,1,n) )+
     &u(ib(m,26),n)*( dk(pix(ib(m,6)),3,j,7,n)
     &+dk(pix(ib(m,5)),4,j,8,n)+
     &dk(pix(ib(m,27)),1,j,5,n)+dk(pix(ib(m,7)),2,j,6,n) )+
     &u(ib(m,27),n)*( dk(pix(ib(m,27)),1,j,1,n)
     &+dk(pix(ib(m,7)),2,j,2,n)+
     &dk(pix(ib(m,6)),3,j,3,n)+dk(pix(ib(m,5)),4,j,4,n)
     &+dk(pix(ib(m,25)),5,j,5,n)+
     &dk(pix(ib(m,15)),6,j,6,n)+dk(pix(ib(m,14)),7,j,7,n)+
     &dk(pix(ib(m,13)),8,j,8,n) )
3000 continue

utot=C
        do 3100 m3=1,3
do 3100 m=1,ns
utot=utot+0.5*u(m,m3)*gb(m,m3)+b(m,m3)*u(m,m3)
gb(m,m3)=gb(m,m3)+b(m,m3)
3100 continue

        return
        end

c  Subroutine that carries out the conjugate gradient relaxation process

      subroutine dembx(ns,Lstep,gg,dk,gtest,ldemb,kkk)
      real gb(8000,3),u(8000,3),dk(100,8,3,8,3)
      real h(8000,3),Ah(8000,3)
      real lambda,gamma
      integer*4 ib(8000,27)
      integer*2 pix(8000)

      common/list3/ib
      common/list4/pix
      common/list6/u
      common/list7/gb
      common/list9/h,Ah

c  Initialize the conjugate direction vector on first call to dembx only
c  For calls to dembx after the first, we want to continue using the
c  value of h determined in the previous call. Of course, if npoints is
c  greater than 1, this initialization step will be run for every new
c  microstructure used, as kkk is reset to 1 every time the counter micro
c  is increased.
      if(kkk.eq.1) then
      do 500 m3=1,3
      do 500 m=1,ns
      h(m,m3)=gb(m,m3)
500   continue
      end if
c  Lstep counts the number of conjugate gradient steps taken in
c  each call to dembx
      Lstep=0

      do 800 ijk=1,ldemb
      Lstep=Lstep+1

      do 290 m3=1,3
      do 290 m=1,ns
      Ah(m,m3)=0.0
290   continue
c  Do global matrix multiply via small stiffness matrices, Ah = A * h
c  The long statement below correctly brings in all the terms from
c  the global matrix A using only the small stiffness matrices dk.
        do 400 j=1,3
        do 400 n=1,3
do 400 m=1,ns
      Ah(m,j)=Ah(m,j)+h(ib(m,1),n)*( dk(pix(ib(m,27)),1,j,4,n)
     &+dk(pix(ib(m,7)),2,j,3,n)
     &+dk(pix(ib(m,25)),5,j,8,n)+dk(pix(ib(m,15)),6,j,7,n) )+
     &h(ib(m,2),n)*( dk(pix(ib(m,27)),1,j,3,n)
     &+dk(pix(ib(m,25)),5,j,7,n) )+
     &h(ib(m,3),n)*( dk(pix(ib(m,27)),1,j,2,n)+dk(pix(ib(m,5)),4,j,3,n)+
     &dk(pix(ib(m,13)),8,j,7,n)+dk(pix(ib(m,25)),5,j,6,n) )+
     &h(ib(m,4),n)*( dk(pix(ib(m,5)),4,j,2,n)
     &+dk(pix(ib(m,13)),8,j,6,n) )+
     &h(ib(m,5),n)*( dk(pix(ib(m,6)),3,j,2,n)+dk(pix(ib(m,5)),4,j,1,n)+
     &dk(pix(ib(m,14)),7,j,6,n)+dk(pix(ib(m,13)),8,j,5,n) )+
     &h(ib(m,6),n)*( dk(pix(ib(m,6)),3,j,1,n)
     &+dk(pix(ib(m,14)),7,j,5,n) )+
     &h(ib(m,7),n)*( dk(pix(ib(m,6)),3,j,4,n)+dk(pix(ib(m,7)),2,j,1,n)+
     &dk(pix(ib(m,14)),7,j,8,n)+dk(pix(ib(m,15)),6,j,5,n) )+
     &h(ib(m,8),n)*( dk(pix(ib(m,7)),2,j,4,n)
     &+dk(pix(ib(m,15)),6,j,8,n) )+
     &h(ib(m,9),n)*( dk(pix(ib(m,25)),5,j,4,n)
     &+dk(pix(ib(m,15)),6,j,3,n) )+
     &h(ib(m,10),n)*( dk(pix(ib(m,25)),5,j,3,n) )+
     &h(ib(m,11),n)*( dk(pix(ib(m,13)),8,j,3,n)
     &+dk(pix(ib(m,25)),5,j,2,n) )+
     &h(ib(m,12),n)*( dk(pix(ib(m,13)),8,j,2,n) )+
     &h(ib(m,13),n)*( dk(pix(ib(m,13)),8,j,1,n)
     &+dk(pix(ib(m,14)),7,j,2,n) )+
     &h(ib(m,14),n)*( dk(pix(ib(m,14)),7,j,1,n) )+
     &h(ib(m,15),n)*( dk(pix(ib(m,14)),7,j,4,n)
     &+dk(pix(ib(m,15)),6,j,1,n) )+
     &h(ib(m,16),n)*( dk(pix(ib(m,15)),6,j,4,n) )+
     &h(ib(m,17),n)*( dk(pix(ib(m,27)),1,j,8,n)
     &+dk(pix(ib(m,7)),2,j,7,n) )+
     &h(ib(m,18),n)*( dk(pix(ib(m,27)),1,j,7,n) )+
     &h(ib(m,19),n)*( dk(pix(ib(m,27)),1,j,6,n)
     &+dk(pix(ib(m,5)),4,j,7,n) )+
     &h(ib(m,20),n)*( dk(pix(ib(m,5)),4,j,6,n) )+
     &h(ib(m,21),n)*( dk(pix(ib(m,5)),4,j,5,n)
     &+dk(pix(ib(m,6)),3,j,6,n) )+
     &h(ib(m,22),n)*( dk(pix(ib(m,6)),3,j,5,n) )+
     &h(ib(m,23),n)*( dk(pix(ib(m,6)),3,j,8,n)
     &+dk(pix(ib(m,7)),2,j,5,n) )+
     &h(ib(m,24),n)*( dk(pix(ib(m,7)),2,j,8,n) )+
     &h(ib(m,25),n)*( dk(pix(ib(m,14)),7,j,3,n)
     &+dk(pix(ib(m,13)),8,j,4,n)+
     &dk(pix(ib(m,15)),6,j,2,n)+dk(pix(ib(m,25)),5,j,1,n) )+
     &h(ib(m,26),n)*( dk(pix(ib(m,6)),3,j,7,n)
     &+dk(pix(ib(m,5)),4,j,8,n)+
     &dk(pix(ib(m,27)),1,j,5,n)+dk(pix(ib(m,7)),2,j,6,n) )+
     &h(ib(m,27),n)*( dk(pix(ib(m,27)),1,j,1,n)
     &+dk(pix(ib(m,7)),2,j,2,n)+
     &dk(pix(ib(m,6)),3,j,3,n)+dk(pix(ib(m,5)),4,j,4,n)
     &+dk(pix(ib(m,25)),5,j,5,n)+
     &dk(pix(ib(m,15)),6,j,6,n)+dk(pix(ib(m,14)),7,j,7,n)+
     &dk(pix(ib(m,13)),8,j,8,n) )
400    continue

        hAh=0.0
```

```
      do 530 m3=1,3
      do 530 m=1,ns
      hAh=hAh+h(m,m3)*Ah(m,m3)
530   continue

      lambda=gg/hAh
      do 540 m3=1,3
      do 540 m=1,ns
      u(m,m3)=u(m,m3)-lambda*h(m,m3)
      gb(m,m3)=gb(m,m3)-lambda*Ah(m,m3)
540   continue

      gglast=gg
      gg=0.0
      do 550 m3=1,3
      do 550 m=1,ns
      gg=gg+gb(m,m3)*gb(m,m3)
550   continue
      if(gg.lt.gtest) goto 1000

      gamma=gg/gglast
      do 570 m3=1,3
      do 570 m=1,ns
      h(m,m3)=gb(m,m3)+gamma*h(m,m3)
570   continue

800   continue

1000  continue

      return
      end

c  Subroutine that computes the six average stresses and six
c  average strains.

      subroutine stress(nx,ny,nz,ns)
      real u(8000,3),gb(8000,3),uu(8,3)
      real dndx(8),dndy(8),dndz(8),es(6,8,3),cmod(100,6,6)
      integer*4 ib(8000,27)
      integer*2 pix(8000)

common/list1/strxx,stryy,strzz,strxz,stryz,strxy
common/list2/exx,eyy,ezz,exz,eyz,exy
common/list3/ib
```

```
        common/list4/pix
common/list6/u
common/list7/gb
        common/list8/cmod
common/list10/sxx,syy,szz,sxz,syz,sxy

      nxy=nx*ny

c  set up single element strain matrix
c  dndx, dndy, and dndz are the components of the average strain
c  matrix in a pixel

      dndx(1)=-0.25
      dndx(2)=0.25
      dndx(3)=0.25
      dndx(4)=-0.25
      dndx(5)=-0.25
      dndx(6)=0.25
      dndx(7)=0.25
      dndx(8)=-0.25
      dndy(1)=-0.25
      dndy(2)=-0.25
      dndy(3)=0.25
      dndy(4)=0.25
      dndy(5)=-0.25
      dndy(6)=-0.25
      dndy(7)=0.25
      dndy(8)=0.25
      dndz(1)=-0.25
      dndz(2)=-0.25
      dndz(3)=-0.25
      dndz(4)=-0.25
      dndz(5)=0.25
      dndz(6)=0.25
      dndz(7)=0.25
      dndz(8)=0.25
c  Build averaged strain matrix, follows code in femat, but for average
c  strain over the pixel, not the strain at a point.
      do 2799 n1=1,6
      do 2799 n2=1,8
      do 2799 n3=1,3
      es(n1,n2,n3)=0.0
2799  continue
      do 2797 n=1,8
      es(1,n,1)=dndx(n)
      es(2,n,2)=dndy(n)
      es(3,n,3)=dndz(n)
      es(4,n,1)=dndz(n)
      es(4,n,3)=dndx(n)
      es(5,n,2)=dndz(n)
      es(5,n,3)=dndy(n)
      es(6,n,1)=dndy(n)
      es(6,n,2)=dndx(n)
2797  continue

c  Compute components of the average stress and strain tensors in each pixel
      strxx=0.0
      stryy=0.0
      strzz=0.0
      strxz=0.0
      stryz=0.0
      strxy=0.0
      sxx=0.0
      syy=0.0
      szz=0.0
      sxz=0.0
      syz=0.0
      sxy=0.0
      do 470 k=1,nz
      do 470 j=1,ny
      do 470 i=1,nx
      m=(k-1)*nxy+(j-1)*nx+i
c  load in elements of 8-vector using pd. bd. conds.
      do 9898 mm=1,3
      uu(1,mm)=u(m,mm)
      uu(2,mm)=u(ib(m,3),mm)
      uu(3,mm)=u(ib(m,2),mm)
      uu(4,mm)=u(ib(m,1),mm)
      uu(5,mm)=u(ib(m,26),mm)
      uu(6,mm)=u(ib(m,19),mm)
      uu(7,mm)=u(ib(m,18),mm)
      uu(8,mm)=u(ib(m,17),mm)
9898  continue
c  Correct for periodic boundary conditions, some displacements are wrong
c  for a pixel on a periodic boundary.  Since they come from an opposite
c  face, need to put in applied strain to correct them.
      if(i.eq.nx) then
      uu(2,1)=uu(2,1)+exx*nx
      uu(2,2)=uu(2,2)+exy*nx
      uu(2,3)=uu(2,3)+exz*nx
      uu(3,1)=uu(3,1)+exx*nx
      uu(3,2)=uu(3,2)+exy*nx
      uu(3,3)=uu(3,3)+exz*nx
      uu(6,1)=uu(6,1)+exx*nx
      uu(6,2)=uu(6,2)+exy*nx
      uu(6,3)=uu(6,3)+exz*nx
      uu(7,1)=uu(7,1)+exx*nx
      uu(7,2)=uu(7,2)+exy*nx
      uu(7,3)=uu(7,3)+exz*nx
      end if
      if(j.eq.ny) then
      uu(3,1)=uu(3,1)+exy*ny
      uu(3,2)=uu(3,2)+eyy*ny
      uu(3,3)=uu(3,3)+eyz*ny
      uu(4,1)=uu(4,1)+exy*ny
      uu(4,2)=uu(4,2)+eyy*ny
      uu(4,3)=uu(4,3)+eyz*ny
      uu(7,1)=uu(7,1)+exy*ny
      uu(7,2)=uu(7,2)+eyy*ny
      uu(7,3)=uu(7,3)+eyz*ny
      uu(8,1)=uu(8,1)+exy*ny
      uu(8,2)=uu(8,2)+eyy*ny
      uu(8,3)=uu(8,3)+eyz*ny
      end if
      if(k.eq.nz) then
      uu(5,1)=uu(5,1)+exz*nz
      uu(5,2)=uu(5,2)+eyz*nz
      uu(5,3)=uu(5,3)+ezz*nz
      uu(6,1)=uu(6,1)+exz*nz
      uu(6,2)=uu(6,2)+eyz*nz
      uu(6,3)=uu(6,3)+ezz*nz
      uu(7,1)=uu(7,1)+exz*nz
      uu(7,2)=uu(7,2)+eyz*nz
      uu(7,3)=uu(7,3)+ezz*nz
      uu(8,1)=uu(8,1)+exz*nz
      uu(8,2)=uu(8,2)+eyz*nz
      uu(8,3)=uu(8,3)+ezz*nz
      end if

c  local stresses and strains in a pixel
      str11=0.0
      str22=0.0
      str33=0.0
      str13=0.0
      str23=0.0
      str12=0.0
      s11=0.0
      s22=0.0
      s33=0.0
      s13=0.0
      s23=0.0
      s12=0.0
      do 465 n3=1,3
      do 465 n8=1,8
      s11=s11+es(1,n8,n3)*uu(n8,n3)
      s22=s22+es(2,n8,n3)*uu(n8,n3)
      s33=s33+es(3,n8,n3)*uu(n8,n3)
      s13=s13+es(4,n8,n3)*uu(n8,n3)
      s23=s23+es(5,n8,n3)*uu(n8,n3)
      s12=s12+es(6,n8,n3)*uu(n8,n3)
      do 465 n=1,6
      str11=str11+cmod(pix(m),1,n)*es(n,n8,n3)*uu(n8,n3)
      str22=str22+cmod(pix(m),2,n)*es(n,n8,n3)*uu(n8,n3)
      str33=str33+cmod(pix(m),3,n)*es(n,n8,n3)*uu(n8,n3)
      str13=str13+cmod(pix(m),4,n)*es(n,n8,n3)*uu(n8,n3)
      str23=str23+cmod(pix(m),5,n)*es(n,n8,n3)*uu(n8,n3)
      str12=str12+cmod(pix(m),6,n)*es(n,n8,n3)*uu(n8,n3)
465   continue
c  sum local strains and stresses into global values
      strxx=strxx+str11
      stryy=stryy+str22
      strzz=strzz+str33
      strxz=strxz+str13
      stryz=stryz+str23
      strxy=strxy+str12
      sxx=sxx+s11
      syy=syy+s22
      szz=szz+s33
      sxz=sxz+s13
      syz=syz+s23
      sxy=sxy+s12
470   continue

c  Volume average of global stresses and strains
      strxx=strxx/float(ns)
      stryy=stryy/float(ns)
      strzz=strzz/float(ns)
      strxz=strxz/float(ns)
      stryz=stryz/float(ns)
      strxy=strxy/float(ns)
      sxx=sxx/float(ns)
      syy=syy/float(ns)
      szz=szz/float(ns)
      sxz=sxz/float(ns)
      syz=syz/float(ns)
      sxy=sxy/float(ns)

      return
      end

c  Subroutine that counts volume fractions

      subroutine assig(ns,nphase,prob)
      integer*2 pix(8000)
      real prob(100)

      common/list4/pix

do 90 i=1,nphase
prob(i)=0.0
90       continue

        do 100 m=1,ns
do 100 i=1,nphase
        if(pix(m).eq.i) then
prob(i)=prob(i)+1
end if
100      continue

do 110 i=1,nphase
prob(i)=prob(i)/float(ns)
110      continue

        return
        end

c  Subroutine that sets up microstructural image

      subroutine ppixel(nx,ny,nz,ns,nphase)
      integer*2 pix(8000)
      common/list4/pix

c  (USER)  If you want to set up a test image inside the program, instead of
c  reading it in from a file, this should be done inside this subroutine.
```

```
      nxy=nx*ny
      do 200 k=1,nz
      do 200 j=1,ny
      do 200 i=1,nx
      m=nxy*(k-1)+nx*(j-1)+i
      read(9,*) pix(m)
200   continue

c  Check for wrong phase labels--less than 1 or greater than nphase
       do 500 m=1,ns
       if(pix(m).lt.1) then
        write(7,*) 'Phase label in pix < 1--error at ',m
       end if
       if(pix(m).gt.nphase) then
        write(7,*) 'Phase label in pix > nphase--error at ',m
       end if
500    continue

      return
      end
```

9.3.3 THERMAL3D.F

```
c  ***********************  thermal3d.f  ***************************
c  BACKGROUND

c  Program adjusts dimensions of unit cell,
c  [(1 + macrostrain) times dimension],
c  in response to phases that have a non-zero eigenstrain and arbitrary
c  elastic moduli tensors.
c  All six macrostrains can adjust their values (3-d program), and are
c  stored in the last two positions in the displacement vector u,
c  as listed below. Periodic boundaries are maintained.
c  In the comments below, (USER) means that this is a section of code
c  that the user might have to change for his particular problem.
c  Therefore the user is encouraged to search for this string.

c  PROBLEM AND VARIABLE DEFINITION

c  The problem being solved is the minimization of the elastic energy
c  1/2 uAu + bu + C + Tu + Y, where b and C are also functions of the
c  macrostrains.
c  The small array zcon computes the thermal strain energy associated
c  with macrostrains (C term), T is the thermal energy term linear in the
c  displacements (built from ss), b is the regular energy term linear in the
c  b is the regular energy term linear in the
c  displacements, u is the displacements including the macrostrains,
c  gb is the energy gradient vector, h,Ah are auxiliary vectors,
c  dk is the single pixel stiffness matrix, pix is the phase
c  identification vector, and ib is the
c  integer matrix for mapping labels from the 1-27 nearest neighbor
c  labelling to the 1-d system labelling.
c  The array prob(i) contains the volume fractions of the i'th phase,
c  strxx, etc. are the six independent (Voigt notation) volume
c  averaged stresses, sxx, etc. are the six independent (Voigt notation)
c  volume averaged strains (not counting the thermal strains).
c  The variable cmod(i,6,6) gives the elastic moduli tensor
c  of the i'th phase, eigen(i,6) gives the six independent elements
c  of the eigenstrain tensor for the i'th phase (Voigt notation)
c  and dk(i,8,3,8,3) is the stiffness matrix of the i'th
c  phase. The parameter nphase gives the number of phases being considered
c  in the problem, and is set by the user.

c  DIMENSIONS

c  The main arrays of the problem, u, gb, h, Ah, b, and T, are dimensioned
```

```
c  as (nx*ny*nz)+2, which is the number of nodal displacements plus two for
c  the macrostrains.
c  Currently the program assumes the number of different phases is
c  100, since phasemod and eigen (the moduli and eigenstrains for each phase)
c  and dk are dimensioned to have at most 100 different kinds.  This
c  is easily changed, by changing the dimension of these three variables
c  throughout the program.  The parameter nphase gives the number of phases
c  All major arrays are passed to subroutines via simple common statements.

c  NOTE ON USE OF PROGRAM

c  Program is set up to allow the macrostrains,
c  which control the overall size of the system, to be dynamic
c  variables, which are adjusted in order to minimize the overall
c  energy.  That means that if there are no eigenstrains specified
c  for any of the phases, the overall strain will always relax to
c  zero.  If it is desired to simply apply a fixed strain, with no
c  eigenstrains, then in subroutines Energy and Dembx, one must
c  zero out the elements of gb (in Energy and in Dembx) that
c  correspond to the macrostrains.  This is easily done.
c  This will fix the gradients of the macrostrains to always to be
c  zero, so that they will not change, so the applied strain (initial
c  values of the macrostrains) will remain fixed.

c  STRONGLY SUGGESTED:  READ MANUAL BEFORE USING PROGRAM!!!

c  (USER)  Change these dimensions and in other subroutines at same
c  time.  For example, search and replace all occurrences throughout
c  the program of "(8002" by "(64000", to go from a 20 x 20 x 20
c  system to a 40 x 40 x 40 system.
        real u(8002,3),gb(8002,3),b(8002,3)
        real h(8002,3),Ah(8002,3),T(8002,3)
real C,dk(100,8,3,8,3)
        real cmod(100,6,6),ss(100,8,3),eigen(100,6)
real zcon(2,3,2,3),pk(6,8,3)
real phasemod(100,2),prob(100)
integer in(27),jn(27),kn(27)
integer*4 ib(8002,27)
integer*2 pix(8002)

        common/list1/strxx,stryy,strzz,strxz,stryz,strxy
        common/list2/h,Ah
common/list3/ib
        common/list4/pix
common/list5/dk,b,C,zcon,Y
```

```
common/list6/u
common/list7/gb
        common/list8/cmod,T,eigen
        common/list10/phasemod,nphase,ss
        common/list11/sxx,syy,szz,sxz,syz,sxy

c  (USER)  Unit 9 is the microstructure input file, unit 7
c  is the results output file.
open (9,file='microstructure.dat')
open (7,file='outputfile.out')

c (USER) nx,ny,nz are the size of the lattice
        nx=20
        ny=20
        nz=20
c  ns=total number of sites
        ns=nx*ny*nz
        write(7,9010) nx,ny,nz,ns
9010  format(' nx= ',i4,'  ny= ',i4,'  nz= ',i4,' ns = ',i8)

c  Add two more entries in the displacement vector for the 6 macrostrains,
c  u(ns+1,1) = exx,u(ns+1,2) = eyy, u(ns+1,3) = ezz,
c  u(nss,1) = exz, u(nss,2) = eyz, u(nss,3) = exy
        nss=ns+2

c  (USER) nphase is the number of phases being considered in the problem.
c  The values of pix(m) will run from 1 to nphase.
nphase=2

c  (USER) gtest is the stopping criterion, compared to gg=gb*gb.
c  If gtest=abc*ns, when gg < gtest, the rms value per pixel
c  of gb is less than sqrt(abc)
        gtest=1.e-20*(nx*ny*nz)
        write(7,*) 'relaxation criterion gtest = ',gtest

c  (USER)
c  The parameter phasemod(i,j) is the bulk (i,1) and shear (i,2) moduli of
c  the i'th phase. These can be
c  input in terms of Young's modulus E (i,1) and Poisson's ratio nu (i,2).
c  The program, in the do 1144 loop, changes them to bulk and shear
c  moduli, using relations for isotropic elastic moduli.
c  For anisotropic moduli tensors, one can directly input the whole tensor
c  cmod in subroutine femat, and skip this part.
c  If you wish to input in terms of bulk (i,1) and shear (i,2) moduli,
c  then simply comment out do 1144 loop.
```

```
        phasemod(1,1)=1.0
        phasemod(1,2)=0.2
phasemod(2,1)=1.0
phasemod(2,2)=0.2

        do 1144 i=1,nphase
        save=phasemod(i,1)
        phasemod(i,1)=phasemod(i,1)/3./(1.-2.*phasemod(i,2))
        phasemod(i,2)=save/2./(1.+phasemod(i,2))
1144    continue

c  (USER) input eigenstrains for each phase
c  (1=xx, 2=yy, 3=zz, 4=xz, 5=yz, 6=xy).
        eigen(1,1)=0.
        eigen(1,2)=0.
        eigen(1,3)=0.
        eigen(1,4)=0.
        eigen(1,5)=0.
        eigen(1,6)=0.
        eigen(2,1)=0.1
        eigen(2,2)=0.
        eigen(2,3)=0.
        eigen(2,4)=0.
        eigen(2,5)=0.
        eigen(2,6)=0.

c  Construct the 27 neighbor table, ib(m,n)

c  First construct the 27 neighbor table in terms of delta i, delta j,
c  and delta k information (see Table 3 in manual)
       in(1)=0
       in(2)=1
       in(3)=1
       in(4)=1
       in(5)=0
       in(6)=-1
       in(7)=-1
       in(8)=-1

       jn(1)=1
       jn(2)=1
       jn(3)=0
       jn(4)=-1
       jn(5)=-1
       jn(6)=-1
```

```
      jn(7)=0
      jn(8)=1

      do 555 n=1,8
      kn(n)=0
      kn(n+8)=-1
      kn(n+16)=1
      in(n+8)=in(n)
      in(n+16)=in(n)
      jn(n+8)=jn(n)
      jn(n+16)=jn(n)
555   continue
      in(25)=0
      in(26)=0
      in(27)=0
      jn(25)=0
      jn(26)=0
      jn(27)=0
      kn(25)=-1
      kn(26)=1
      kn(27)=0
c  Now construct neighbor table according to 1-d labels
c  Matrix ib(m,n) gives the 1-d label of the n'th neighbor (n=1,27) of
c  the node labelled m.
      nxy=nx*ny
      do 1020 k=1,nz
      do 1020 j=1,ny
      do 1020 i=1,nx
      m=nxy*(k-1)+nx*(j-1)+i
      do 1004 n=1,27
      i1=i+in(n)
      j1=j+jn(n)
      k1=k+kn(n)
      if(i1.lt.1) i1=i1+nx
      if(i1.gt.nx) i1=i1-nx
      if(j1.lt.1) j1=j1+ny
      if(j1.gt.ny) j1=j1-ny
      if(k1.lt.1) k1=k1+nz
      if(k1.gt.nz) k1=k1-nz
      m1=nxy*(k1-1)+nx*(j1-1)+i1
      ib(m,n)=m1
1004  continue
1020  continue

c  Compute the average stress and strain, as well as the macrostrains (overall
c  system size and shape) in each microstructure.
c  (USER) npoints is the number of microstructures to use.
        npoints=1

        do 8000 micro=1,npoints
c  Read in a microstructure in subroutine ppixel, and set up pix(m)
c  with the appropriate phase assignments.
        call ppixel(nx,ny,nz,ns,nphase)
c  Count and output the volume fractions of the different phases
        call assig(ns,nphase,prob)
do 8050 i=1,nphase
write(7,9065) i,prob(i)
9065 format(' Volume fraction of phase ',i3,'  is ',f10.8)
8050 continue
c  output elastic moduli (bulk and shear) for each phase
        write(7,*) '  Phase Moduli'
        do 111 i=1,nphase
        write(7,9020) i,phasemod(i,1),phasemod(i,2)
9020 format(' Phase ',i3,' bulk = ',f12.6,' shear = ',f12.6)
111 continue
c  output thermal strains for each phase
        write(7,*) '  Thermal Strains'
        do 119 i=1,nphase
        write(7,9029) i,eigen(i,1),eigen(i,2),eigen(i,3)
        write(7,9029) i,eigen(i,4),eigen(i,5),eigen(i,6)
9029    format('Phase ',i3,'  ',3f6.2)
119 continue
c  (USER) Set inital macrostrains of computational cell
      u(ns+1,1)=0.0
      u(ns+1,2)=0.0
      u(ns+1,3)=0.0
      u(nss,1)=0.0
      u(nss,2)=0.0
      u(nss,3)=0.0
c Apply homogeneous macroscopic strain as the initial condition
c to displacement variables
do 1050 k=1,nz
do 1050 j=1,ny
        do 1050 i=1,nx
m=nxy*(k-1)+nx*(j-1)+i
x=float(i-1)
y=float(j-1)
z=float(k-1)
                u(m,1)=x*u(ns+1,1)+y*u(nss,3)+z*u(nss,1)
                u(m,2)=x*u(nss,3)+y*u(ns+1,2)+z*u(nss,2)
                u(m,3)=x*u(nss,1)+y*u(nss,2)+z*u(ns+1,3)
1050 continue

c  Set up the finite element stiffness matrices,the constant, C,
c  the vector, b, required for the energy. b and C depend on the macrostrains.
c  When they are updated, the values of b and C are updated too via
c  calling subroutine femat.
c  Only compute the thermal strain terms the first time femat is called,
c  (iskip=0) as they are unaffected by later changes (iskip=1) in
c  displacements and macrostrains.
c  Compute initial value of gradient gb and gg=gb*gb.
        iskip=0
call femat(nx,ny,nz,ns,iskip)
        call energy(nx,ny,nz,ns,utot)
  gg=0.0
        do 100 m3=1,3
        do 100 m=1,nss
        gg=gg+gb(m,m3)*gb(m,m3)
100     continue
write(7,9042) utot,gg
9042 format(' energy = ',e15.8,'  gg=  ',e15.8)
        call flush(7)

c  Relaxation loop
c  (USER) kmax is the maximum number of times that dembx will be called,
c  with ldemb conjugate gradient steps performed during each call.
c  The total number of conjugate gradient steps allowed for a given elastic
c  computation is kmax*ldemb.
        kmax=40
        ldemb=50
        ltot=0

        do 5000 kkk=1,kmax

c  Call dembx to implement conjugate gradient routine
        write(7,*) 'Going into dembx, call no. ',kkk
        call dembx(nx,ny,nz,ns,Lstep,gg,gtest,ldemb,kkk)
        ltot=ltot+Lstep
c  Call energy to compute energy after dembx call. If gg < gtest, this
c  will be the final energy.  If gg is still larger than gtest, then this
c  will give an intermediate energy with which to check how the
c  relaxation process is coming along. The call to energy does not
c  change the gradient or the value of gg.
c  Need to first call femat to update the vector b, as the value of the
c  components of b depend on the macrostrains.
```

```
        iskip=1
call femat(nx,ny,nz,ns,iskip)
        call energy(nx,ny,nz,ns,utot)
write(7,9043) utot,gg,ltot
9043    format(' energy = ',e15.8,' gg=  ',e15.8,' ltot = ',i6)
        call flush(7)

c  If relaxation process is finished, jump out of loop
        if(gg.lt.gtest) goto 444
c  Output stresses, strains, and macrostrains as an additional aid in judging
c  how well the relaxation process is proceeding.
      call stress(nx,ny,nz,ns)
      write(7,*) ' stresses:  xx,yy,zz,xz,yz,xy'
      write(7,*) strxx,stryy,strzz,strxz,stryz,strxy
      write(7,*) ' strains:  xx,yy,zz,xz,yz,xy'
      write(7,*) sxx,syy,szz,sxz,syz,sxy

      write(7,*) ' macrostrains in same order'
      write(7,*) u(ns+1,1),u(ns+1,2),u(ns+1,3)
      write(7,*) u(nss,1),u(nss,2),u(nss,3)
      write(7,*) 'avg = ',(u(ns+1,1)+u(ns+1,2)+u(ns+1,3))/3.

5000    continue

444   call stress(nx,ny,nz,ns)
      write(7,*) ' stresses:  xx,yy,zz,xz,yz,xy'
      write(7,*) strxx,stryy,strzz,strxz,stryz,strxy
      write(7,*) ' strains:  xx,yy,zz,xz,yz,xy'
      write(7,*) sxx,syy,szz,sxz,syz,sxy

      write(7,*) ' macrostrains in same order'
      write(7,*) u(ns+1,1),u(ns+1,2),u(ns+1,3)
      write(7,*) u(nss,1),u(nss,2),u(nss,3)
      write(7,*) 'avg = ',(u(ns+1,1)+u(ns+1,2)+u(ns+1,3))/3.

8000  continue

        end

c  Subroutine sets up the stiffness matrices, the linear term in the
c  regular displacements, b, and the constant term, C, which come from
c  the periodic boundary conditions, the term linear in the displacments,
c  T, that comes from the thermal strains, and the constant term Y.

      subroutine femat(nx,ny,nz,ns,iskip)
      real u(8002,3),b(8002,3),T(8002,3)
      real dk(100,8,3,8,3),phasemod(100,2),dndx(8),dndy(8),dndz(8)
      real g(3,3,3),econ,ck(6,6),cmu(6,6),cmod(100,6,6)
      real es(6,8,3),zcon(2,3,2,3),ss(100,8,3)
      real eigen(100,6),delta(8,3)
      integer is(8),iskip
      integer*4 ib(8002,27)
      integer*2 pix(8002)

common/list3/ib
        common/list4/pix
common/list5/dk,b,C,zcon,Y
common/list6/u
        common/list8/cmod,T,eigen
        common/list10/phasemod,nphase,ss

      nxy=nx*ny

c  Generate dk, zcon, T, and Y on first pass. After that they are
c  constant, snce they are independent of the macrostrains.  Only b gets
c  upgraded as the macrostrains change.
c  Line number 1221 is the routine for b.
      if(iskip.eq.1) goto 1221

c  initialize stiffness matrices
      do 40 m=1,nphase
      do 40 l=1,3
      do 40 k=1,3
      do 40 j=1,8
      do 40 i=1,8
      dk(m,i,k,j,l)=0.0
40    continue
c  initialize zcon matrix (gives C term for arbitrary macrostrains)
      do 42 i=1,2
      do 42 j=1,2
      do 42 mi=1,3
      do 42 mj=1,3
      zcon(i,mi,j,mj)=0.0
42    continue
c  (USER) An anisotropic elastic moduli tensor could be input at this point,
c  bypassing this part, which assumes isotropic elasticity, so that there
c  are only two independent numbers making up the elastic moduli tensor,
c  the bulk modulus K and the shear modulus G.

c  Set up elastic moduli matrices for each kind of element
```

```
c  ck and cmu are the bulk modulus and shear modulus matrices, which
c  need to multiplied by the actual bulk and shear moduli in each phase.

      ck(1,1)=1.0
      ck(1,2)=1.0
      ck(1,3)=1.0
      ck(1,4)=0.0
      ck(1,5)=0.0
      ck(1,6)=0.0
      ck(2,1)=1.0
      ck(2,2)=1.0
      ck(2,3)=1.0
      ck(2,4)=0.0
      ck(2,5)=0.0
      ck(2,6)=0.0
      ck(3,1)=1.0
      ck(3,2)=1.0
      ck(3,3)=1.0
      ck(3,4)=0.0
      ck(3,5)=0.0
      ck(3,6)=0.0
      ck(4,1)=0.0
      ck(4,2)=0.0
      ck(4,3)=0.0
      ck(4,4)=0.0
      ck(4,5)=0.0
      ck(4,6)=0.0
      ck(5,1)=0.0
      ck(5,2)=0.0
      ck(5,3)=0.0
      ck(5,4)=0.0
      ck(5,5)=0.0
      ck(5,6)=0.0
      ck(6,1)=0.0
      ck(6,2)=0.0
      ck(6,3)=0.0
      ck(6,4)=0.0
      ck(6,5)=0.0
      ck(6,6)=0.0

      cmu(1,1)=4.0/3.0
      cmu(1,2)=-2.0/3.0
      cmu(1,3)=-2.0/3.0
      cmu(1,4)=0.0
      cmu(1,5)=0.0
      cmu(1,6)=0.0
      cmu(2,1)=-2.0/3.0
      cmu(2,2)=4.0/3.0
      cmu(2,3)=-2.0/3.0
      cmu(2,4)=0.0
      cmu(2,5)=0.0
      cmu(2,6)=0.0
      cmu(3,1)=-2.0/3.0
      cmu(3,2)=-2.0/3.0
      cmu(3,3)=4.0/3.0
      cmu(3,4)=0.0
      cmu(3,5)=0.0
      cmu(3,6)=0.0
      cmu(4,1)=0.0
      cmu(4,2)=0.0
      cmu(4,3)=0.0
      cmu(4,4)=1.0
      cmu(4,5)=0.0
      cmu(4,6)=0.0
      cmu(5,1)=0.0
      cmu(5,2)=0.0
      cmu(5,3)=0.0
      cmu(5,4)=0.0
      cmu(5,5)=1.0
      cmu(5,6)=0.0
      cmu(6,1)=0.0
      cmu(6,2)=0.0
      cmu(6,3)=0.0
      cmu(6,4)=0.0
      cmu(6,5)=0.0
      cmu(6,6)=1.0

      do 31 k=1,nphase
      do 21 j=1,6
      do 11 i=1,6
      cmod(k,i,j)=phasemod(k,1)*ck(i,j)+phasemod(k,2)*cmu(i,j)
11    continue
21    continue
31    continue
c  Set up Simpson's integration rule weight vector
      do 30 k=1,3
      do 30 j=1,3
      do 30 i=1,3
      nm=0
      if(i.eq.2) nm=nm+1
      if(j.eq.2) nm=nm+1
      if(k.eq.2) nm=nm+1
      g(i,j,k)=4.0**nm
30    continue

c  Loop over the nphase kinds of pixels and
c  Simpson's rule quadrature points in order to compute the stiffness
c  matrices.  Stiffness matrices of trilinear finite elements are quadratic
c  in x, y, and z, so that Simpson's rule quadrature gives exact results.
      do 4000 ijk=1,nphase
      do 3000 k=1,3
      do 3000 j=1,3
      do 3000 i=1,3
      x=float(i-1)/2.0
      y=float(j-1)/2.0
      z=float(k-1)/2.0
c  dndx means the negative derivative with respect to x, of the shape
c  matrix N (see manual, Sec. 2.2), dndy and dndz are similar.
      dndx(1)=-(1.0-y)*(1.0-z)
      dndx(2)=(1.0-y)*(1.0-z)
      dndx(3)=y*(1.0-z)
      dndx(4)=-y*(1.0-z)
      dndx(5)=-(1.0-y)*z
      dndx(6)=(1.0-y)*z
      dndx(7)=y*z
      dndx(8)=-y*z
      dndy(1)=-(1.0-x)*(1.0-z)
      dndy(2)=-x*(1.0-z)
      dndy(3)=x*(1.0-z)
      dndy(4)=(1.0-x)*(1.0-z)
      dndy(5)=-(1.0-x)*z
      dndy(6)=-x*z
      dndy(7)=x*z
      dndy(8)=(1.0-x)*z
      dndz(1)=-(1.0-x)*(1.0-y)
      dndz(2)=-x*(1.0-y)
      dndz(3)=-x*y
      dndz(4)=-(1.0-x)*y
      dndz(5)=(1.0-x)*(1.0-y)
      dndz(6)=x*(1.0-y)
      dndz(7)=x*y
      dndz(8)=(1.0-x)*y
c  now build strain matrix
      do 2799 n1=1,6
      do 2799 n2=1,8
      do 2799 n3=1,3
      es(n1,n2,n3)=0.0
2799  continue
      do 2797 n=1,8
      es(1,n,1)=dndx(n)
      es(2,n,2)=dndy(n)
      es(3,n,3)=dndz(n)
      es(4,n,1)=dndz(n)
      es(4,n,3)=dndx(n)
      es(5,n,2)=dndz(n)
      es(5,n,3)=dndy(n)
      es(6,n,1)=dndy(n)
      es(6,n,2)=dndx(n)
2797  continue
c  now do matrix multiply to determine value at (x,y,z), multiply by
c  proper weight, and sum into dk, the stiffness matrix
      do 900 mm=1,3
      do 900 nn=1,3
      do 900 ii=1,8
      do 900 jj=1,8
c  define sum over strain matrices and elastic moduli matrix for
c  stiffness matrix
      sum=0.0
      do 890 kk=1,6
      do 890 ll=1,6
      sum=sum+es(kk,ii,mm)*cmod(ijk,kk,ll)*es(ll,jj,nn)
890   continue
      dk(ijk,ii,mm,jj,nn)=dk(ijk,ii,mm,jj,nn)+g(i,j,k)*sum/216.
900   continue
3000  continue
4000  continue

c  Now compute the ss matrices, which give the thermal strain terms
c  for the i'th phase, single pixel.

      dndx(1)=-0.25
      dndx(2)=0.25
      dndx(3)=0.25
      dndx(4)=-0.25
      dndx(5)=-0.25
      dndx(6)=0.25
      dndx(7)=0.25
      dndx(8)=-0.25
      dndy(1)=-0.25
      dndy(2)=-0.25
      dndy(3)=0.25
      dndy(4)=0.25
      dndy(5)=-0.25
      dndy(6)=-0.25
      dndy(7)=0.25
      dndy(8)=0.25
      dndz(1)=-0.25
      dndz(2)=-0.25
      dndz(3)=-0.25
      dndz(4)=-0.25
      dndz(5)=0.25
      dndz(6)=0.25
      dndz(7)=0.25
      dndz(8)=0.25
c  now build average strain matrix
      do 3799 n1=1,6
      do 3799 n2=1,8
      do 3799 n3=1,3
      es(n1,n2,n3)=0.0
3799  continue
      do 3797 n=1,8
      es(1,n,1)=dndx(n)
      es(2,n,2)=dndy(n)
      es(3,n,3)=dndz(n)
      es(4,n,1)=dndz(n)
      es(4,n,3)=dndx(n)
      es(5,n,2)=dndz(n)
      es(5,n,3)=dndy(n)
      es(6,n,1)=dndy(n)
      es(6,n,2)=dndx(n)
3797  continue
      do 3598 mmm=1,nphase
      do 3798 nn=1,3
      do 3798 mm=1,8
      sum=0.0
      do 3698 nm=1,6
      do 3698 n=1,6
      sum=sum+cmod(mmm,n,nm)*es(n,mm,nn)*eigen(mmm,nm)
3698  continue
      ss(mmm,mm,nn)=sum
3798  continue
3598  continue

c  now call subroutine const to generate zcon
c  zcon is a (2,3) x (2,3) matrix
```

```
      call const(dk,ns,zcon,nx,ny,nz)

c  Now set up linear term, T, for thermal energy. It does not depend
c  on the macrostrains or displacements, so there is no need to update it
c  as the macrostrains change. T is built up out of the ss matrices.

      nss=ns+2
      do 6066 m3=1,3
      do 6066 m=1,nss
      T(m,m3)=0.0
6066  continue

c  For all cases, the correspondence between 1-8 finite element node
c  labels and the 1-27 neighbor labels is (see Table 4 in manual):
c  1:ib(m,27), 2:ib(m,3),3:ib(m,2),4:ib(m,1)
c  5:ib(m,26),6:ib(m,19),7:ib(m,18),8:ib(m,17)
      is(1)=27
      is(2)=3
      is(3)=2
      is(4)=1
      is(5)=26
      is(6)=19
      is(7)=18
      is(8)=17
c  Do all points, but no macrostrain terms
c  note:  factor of 2 on linear thermal term is cancelled
c  by factor of 1/2 out in front of total energy term
      do 6601 k=1,nz
      do 6601 j=1,ny
      do 6601 i=1,nx
      m=nxy*(k-1)+nx*(j-1)+i
      do 6600 mm=1,8
      do 6600 nn=1,3
      T(ib(m,is(mm)),nn)=T(ib(m,is(mm)),nn)-ss(pix(m),mm,nn)
6600  continue
6601  continue

c  now need to pick up and sum in all terms multiplying macrostrains
      do 7788 ipp=1,2
      do 7788 jpp=1,3
      exx=0.0
      eyy=0.0
      ezz=0.0
      exz=0.0
      eyz=0.0
      exy=0.0
      if(ipp.eq.1.and.jpp.eq.1) exx=1.0
      if(ipp.eq.1.and.jpp.eq.2) eyy=1.0
      if(ipp.eq.1.and.jpp.eq.3) ezz=1.0
      if(ipp.eq.2.and.jpp.eq.1) exz=1.0
      if(ipp.eq.2.and.jpp.eq.2) eyz=1.0
      if(ipp.eq.2.and.jpp.eq.3) exy=1.0

c  x=nx face
      do 6001 i3=1,3
      do 6001 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
6001  continue

      do 6000 j=1,ny-1
      do 6000 k=1,nz-1
      m=nxy*(k-1)+j*nx
      do 6900 nn=1,3
      do 6900 mm=1,8
      T(ns+ipp,jpp)=T(ns+ipp,jpp)-ss(pix(m),mm,nn)*delta(mm,nn)
6900  continue
6000  continue

c  y=ny face
      do 6011 i3=1,3
      do 6011 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
6011  continue
      do 6010 i=1,nx-1
      do 6010 k=1,nz-1
      m=nxy*(k-1)+nx*(ny-1)+i
      do 6901 nn=1,3
      do 6901 mm=1,8
      T(ns+ipp,jpp)=T(ns+ipp,jpp)-ss(pix(m),mm,nn)*delta(mm,nn)
6901  continue
```

```
6010  continue
c  z=nz face
      do 6021 i3=1,3
      do 6021 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
6021  continue
      do 6020 i=1,nx-1
      do 6020 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      do 6902 nn=1,3
      do 6902 mm=1,8
      T(ns+ipp,jpp)=T(ns+ipp,jpp)-ss(pix(m),mm,nn)*delta(mm,nn)
6902  continue
6020  continue
c  x=nx y=ny edge
      do 6031 i3=1,3
      do 6031 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.6) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      delta(i8,1)=exy*ny+exx*nx
      delta(i8,2)=eyy*ny+exy*nx
      delta(i8,3)=eyz*ny+exz*nx
      end if
6031  continue
      do 6030 k=1,nz-1
      m=nxy*k
      do 6903 nn=1,3
      do 6903 mm=1,8
      T(ns+ipp,jpp)=T(ns+ipp,jpp)-ss(pix(m),mm,nn)*delta(mm,nn)
6903  continue
```

```
6030  continue
c  x=nx z=nz edge
      do 6041 i3=1,3
      do 6041 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.5.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exz*nz+exx*nx
      delta(i8,2)=eyz*nz+exy*nx
      delta(i8,3)=ezz*nz+exz*nx
      end if
6041  continue
      do 6040 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      do 6904 nn=1,3
      do 6904 mm=1,8
      T(ns+ipp,jpp)=T(ns+ipp,jpp)-ss(pix(m),mm,nn)*delta(mm,nn)
6904  continue
6040  continue
c  y=ny z=nz edge
      do 6051 i3=1,3
      do 6051 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.3.or.i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
6051  continue
      do 6050 i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      do 6905 nn=1,3
      do 6905 mm=1,8
      T(ns+ipp,jpp)=T(ns+ipp,jpp)-ss(pix(m),mm,nn)*delta(mm,nn)
6905  continue
6050  continue
c  x=nx y=ny z=nz corner
      do 6061 i3=1,3
      do 6061 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.5) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      if(i8.eq.6) then
      delta(i8,1)=exx*nx+exz*nz
      delta(i8,2)=exy*nx+eyz*nz
      delta(i8,3)=exz*nx+ezz*nz
      end if
      if(i8.eq.3) then
      delta(i8,1)=exx*nx+exy*ny
      delta(i8,2)=exy*nx+eyy*ny
      delta(i8,3)=exz*nx+eyz*ny
      end if
      if(i8.eq.7) then
      delta(i8,1)=exx*nx+exy*ny+exz*nz
      delta(i8,2)=exy*nx+eyy*ny+eyz*nz
      delta(i8,3)=exz*nx+eyz*ny+ezz*nz
      end if
6061  continue
      m=nx*ny*nz
      do 6906 mm=1,8
      do 6906 nn=1,3
      T(ns+ipp,jpp)=T(ns+ipp,jpp)-ss(pix(m),mm,nn)*delta(mm,nn)
6906  continue
7788  continue
c  now compute Y, the 0.5(eigen)Cij(eigen) energy, doesn't ever change
c  with macrostrain or displacements
      Y=0.0
      do 8811 m=1,ns
      do 8811 n=1,6
      do 8811 nn=1,6
      Y=Y+0.5*eigen(pix(m),n)*cmod(pix(m),n,nn)*eigen(pix(m),nn)
8811  continue

c  Following needs to be run after every change in macrostrain
c  when energy is recomputed.

1221  continue
c  Use auxiliary variables (exx, etc.) instead of u() variable, for
c  convenience, and to make the following code easier to read.
      exx=u(ns+1,1)
      eyy=u(ns+1,2)
      ezz=u(ns+1,3)
      exz=u(nss,1)
      eyz=u(nss,2)
      exy=u(nss,3)
c  Now set up vector for linear term that comes from periodic boundary
c  conditions.  Notation and conventions same as for T term.
c  This is done using the stiffness matrices, and the periodic terms
c  in the macrostrains.  It is easier to set up b this way than to
c  analytically write out all the terms involved.

      do 5000 m3=1,3
      do 5000 m=1,ns
      b(m,m3)=0.0
5000  continue
c  For all cases, the correspondence between 1-8 finite element node
c  labels and the 1-27 neighbor labels is (see Table 4 in manual):
c  1:ib(m,27), 2:ib(m,3),3:ib(m,2),4:ib(m,1)
```

```
c  5:ib(m,26),6:ib(m,19),7:ib(m,18),8:ib(m,17)
      is(1)=27
      is(2)=3
      is(3)=2
      is(4)=1
      is(5)=26
      is(6)=19
      is(7)=18
      is(8)=17

      C=0.0
c  x=nx face
      do 2001 i3=1,3
      do 2001 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
2001  continue

      do 2000 j=1,ny-1
      do 2000 k=1,nz-1
      m=nxy*(k-1)+j*nx
      do 1900 nn=1,3
      do 1900 mm=1,8
      sum=0.0
      do 1899 m3=1,3
      do 1899 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
1899  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1900  continue
2000  continue
c  y=ny face
      do 2011 i3=1,3
      do 2011 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
2011  continue
```

```
      do 2010 i=1,nx-1
      do 2010 k=1,nz-1
      m=nxy*(k-1)+nx*(ny-1)+i
      do 1901 nn=1,3
      do 1901 mm=1,8
      sum=0.0
      do 2099 m3=1,3
      do 2099 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2099  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1901  continue
2010  continue
c  z=nz face
      do 2021 i3=1,3
      do 2021 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
2021  continue
      do 2020 i=1,nx-1
      do 2020 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      do 1902 nn=1,3
      do 1902 mm=1,8
      sum=0.0
      do 2019 m3=1,3
      do 2019 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2019  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1902  continue
2020  continue
c  x=nx y=ny edge
      do 2031 i3=1,3
      do 2031 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.6) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      delta(i8,1)=exy*ny+exx*nx
      delta(i8,2)=eyy*ny+exy*nx
      delta(i8,3)=eyz*ny+exz*nx
      end if
2031  continue
      do 2030 k=1,nz-1
      m=nxy*k
      do 1903 nn=1,3
      do 1903 mm=1,8
      sum=0.0
      do 2029 m3=1,3
      do 2029 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2029  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1903  continue
2030  continue
c  x=nx z=nz edge
      do 2041 i3=1,3
      do 2041 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.5.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exz*nz+exx*nx
      delta(i8,2)=eyz*nz+exy*nx
      delta(i8,3)=ezz*nz+exz*nx
      end if
2041  continue
      do 2040 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      do 1904 nn=1,3
      do 1904 mm=1,8
      sum=0.0
      do 2039 m3=1,3
      do 2039 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2039  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1904  continue
2040  continue
c  y=ny z=nz edge
      do 2051 i3=1,3
      do 2051 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.3.or.i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
2051  continue
      do 2050 i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      do 1905 nn=1,3
      do 1905 mm=1,8
      sum=0.0
      do 2049 m3=1,3
      do 2049 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2049  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1905  continue
2050  continue
c  x=nx y=ny z=nz corner
      do 2061 i3=1,3
      do 2061 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.5) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      if(i8.eq.6) then
      delta(i8,1)=exx*nx+exz*nz
      delta(i8,2)=exy*nx+eyz*nz
      delta(i8,3)=exz*nx+ezz*nz
      end if
      if(i8.eq.3) then
      delta(i8,1)=exx*nx+exy*ny
      delta(i8,2)=exy*nx+eyy*ny
      delta(i8,3)=exz*nx+eyz*ny
      end if
      if(i8.eq.7) then
      delta(i8,1)=exx*nx+exy*ny+exz*nz
      delta(i8,2)=exy*nx+eyy*ny+eyz*nz
      delta(i8,3)=exz*nx+eyz*ny+ezz*nz
      end if
2061  continue
      m=nx*ny*nz
      do 1906 nn=1,3
      do 1906 mm=1,8
      sum=0.0
      do 2059 m3=1,3
      do 2059 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2059  continue
```

```
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1906  continue

      return
      end

c  Subroutine that computes derivatives of the b-vector with respect
c  to the macrostrains.  Since b is linear in the macrostrains, the
c  derivative with respect to any one of them can be computed simply
c  by letting that macrostrain, within the subroutine, be equal to one,
c  and all the other macrostrains to be zero.
c  Very similar to 1221 loop in femat for b.

      subroutine bgrad(nx,ny,nz,ns,exx,eyy,ezz,exz,eyz,exy)
      real b(8002,3)
      real dk(100,8,3,8,3),delta(8,3),zcon(2,3,2,3)
      integer is(8)
      integer*4 ib(8002,27)
      integer*2 pix(8002)

      common/list3/ib
      common/list4/pix
      common/list5/dk,b,C,zcon,Y

      nxy=nx*ny
c  exx, eyy, ezz, exz, eyz, exy are the artificial macrostrains used
c  to get the gradient terms (appropriate combinations of 1's and 0's).

c  Set up vector for linear term

      do 5000 m3=1,3
      do 5000 m=1,ns
      b(m,m3)=0.0
5000  continue
      is(1)=27
      is(2)=3
      is(3)=2
      is(4)=1
      is(5)=26
      is(6)=19
      is(7)=18
      is(8)=17

c  x=nx face
      do 2001 i3=1,3
      do 2001 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
2001  continue

      do 2000 j=1,ny-1
      do 2000 k=1,nz-1
      m=nxy*(k-1)+j*nx
      do 1900 nn=1,3
      do 1900 mm=1,8
      sum=0.0
      do 1899 m3=1,3
      do 1899 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
1899  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1900  continue
2000  continue
c  y=ny face
      do 2011 i3=1,3
      do 2011 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
2011  continue
      do 2010 i=1,nx-1
      do 2010 k=1,nz-1
      m=nxy*(k-1)+nx*(ny-1)+i
      do 1901 nn=1,3
      do 1901 mm=1,8
      sum=0.0
      do 2099 m3=1,3
      do 2099 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2099  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1901  continue
2010  continue
```

```
c  z=nz face
      do 2021 i3=1,3
      do 2021 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
2021  continue
      do 2020 i=1,nx-1
      do 2020 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      do 1902 nn=1,3
      do 1902 mm=1,8
      sum=0.0
      do 2019 m3=1,3
      do 2019 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2019  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1902  continue
2020  continue
c  x=nx y=ny edge
      do 2031 i3=1,3
      do 2031 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.6) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      delta(i8,1)=exy*ny+exx*nx
      delta(i8,2)=eyy*ny+exy*nx
      delta(i8,3)=eyz*ny+exz*nx
      end if
2031  continue
      do 2030 k=1,nz-1
      m=nxy*k
      do 1903 nn=1,3
      do 1903 mm=1,8
      sum=0.0
      do 2029 m3=1,3
      do 2029 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2029  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1903  continue
2030  continue
c  x=nx z=nz edge
      do 2041 i3=1,3
      do 2041 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.5.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exz*nz+exx*nx
      delta(i8,2)=eyz*nz+exy*nx
      delta(i8,3)=ezz*nz+exz*nx
      end if
2041  continue
      do 2040 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      do 1904 nn=1,3
      do 1904 mm=1,8
      sum=0.0
      do 2039 m3=1,3
      do 2039 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2039  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1904  continue
2040  continue
c  y=ny z=nz edge
      do 2051 i3=1,3
      do 2051 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.3.or.i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
2051  continue
      do 2050 i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      do 1905 nn=1,3
      do 1905 mm=1,8
      sum=0.0
      do 2049 m3=1,3
      do 2049 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2049  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1905  continue
2050  continue
c  x=nx y=ny z=nz corner
      do 2061 i3=1,3
      do 2061 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.5) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      if(i8.eq.6) then
      delta(i8,1)=exx*nx+exz*nz
      delta(i8,2)=exy*nx+eyz*nz
      delta(i8,3)=exz*nx+ezz*nz
      end if
      if(i8.eq.3) then
      delta(i8,1)=exx*nx+exy*ny
      delta(i8,2)=exy*nx+eyy*ny
      delta(i8,3)=exz*nx+eyz*ny
      end if
      if(i8.eq.7) then
      delta(i8,1)=exx*nx+exy*ny+exz*nz
      delta(i8,2)=exy*nx+eyy*ny+eyz*nz
      delta(i8,3)=exz*nx+eyz*ny+ezz*nz
      end if
2061  continue
      m=nx*ny*nz
      do 1906 nn=1,3
      do 1906 mm=1,8
      sum=0.0
      do 2059 m3=1,3
      do 2059 m8=1,8
      sum=sum+delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)
2059  continue
      b(ib(m,is(mm)),nn)=b(ib(m,is(mm)),nn)+sum
1906  continue
      return
      end

c  Subroutine computes the quadratic term in the macrostrains, that comes
c  from the periodic boundary conditions, and sets it up as a
c  (2,3) x (2,3) matrix that couples to the six macrostrains

      subroutine const(dk,ns,zcon,nx,ny,nz)
      real dk(100,8,3,8,3),zcon(2,3,2,3),delta(8,3)
      real pp(6,6),s(6,6)
      integer*2 pix(8002)
      common/list4/pix

c  routine to set up 6 x 6 matrix for energy term involving macro-strains
c  only, pulled out of femat
c  Idea is to evaluate the quadratic term in the macrostrains repeatedly
c  for choices of strain like
c  exx=1, exy=1, all others = 0, build up 21 choices, then recombine
c  to get matrix elements by themselves

      nxy=nx*ny
      nss=ns+2

      do 1111 i=1,6
      do 1111 j=1,6
      s(i,j)=0.0
      pp(i,j)=0.0
1111  continue

      do 5000 ii=1,6
      do 5000 jj=ii,6
      econ=0.0
      exx=0.0
      eyy=0.0
      ezz=0.0
      exz=0.0
      eyz=0.0
      exy=0.0
      if(ii.eq.1.and.jj.eq.1) exx=1.0
      if(ii.eq.2.and.jj.eq.2) eyy=1.0
      if(ii.eq.3.and.jj.eq.3) ezz=1.0
      if(ii.eq.4.and.jj.eq.4) exz=1.0
      if(ii.eq.5.and.jj.eq.5) eyz=1.0
      if(ii.eq.6.and.jj.eq.6) exy=1.0
      if(ii.eq.1.and.jj.eq.2) then
      exx=1.0
      eyy=1.0
      end if
      if(ii.eq.1.and.jj.eq.3) then
      exx=1.0
      ezz=1.0
      end if
      if(ii.eq.1.and.jj.eq.4) then
      exx=1.0
      exz=1.0
      end if
```

```
if(ii.eq.1.and.jj.eq.5) then
exx=1.0
eyz=1.0
end if
if(ii.eq.1.and.jj.eq.6) then
exx=1.0
exy=1.0
end if
if(ii.eq.2.and.jj.eq.3) then
eyy=1.0
ezz=1.0
end if
if(ii.eq.2.and.jj.eq.4) then
eyy=1.0
exz=1.0
end if
if(ii.eq.2.and.jj.eq.5) then
eyy=1.0
eyz=1.0
end if
if(ii.eq.2.and.jj.eq.6) then
eyy=1.0
exy=1.0
end if
if(ii.eq.3.and.jj.eq.4) then
ezz=1.0
exz=1.0
end if
if(ii.eq.3.and.jj.eq.5) then
ezz=1.0
eyz=1.0
end if
if(ii.eq.3.and.jj.eq.6) then
ezz=1.0
exy=1.0
end if
if(ii.eq.4.and.jj.eq.5) then
exz=1.0
eyz=1.0
end if
if(ii.eq.4.and.jj.eq.6) then
exz=1.0
exy=1.0
end if
if(ii.eq.5.and.jj.eq.6) then
```

```
      eyz=1.0
      exy=1.0
      end if
c  x=nx face
      do 2001 i3=1,3
      do 2001 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
2001  continue

      do 2000 j=1,ny-1
      do 2000 k=1,nz-1
      m=nxy*(k-1)+j*nx
      do 1900 nn=1,3
      do 1900 mm=1,8
      do 1899 m3=1,3
      do 1899 m8=1,8
      econ=econ+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
1899  continue
1900  continue
2000  continue
c  y=ny face
      do 2011 i3=1,3
      do 2011 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
2011  continue
      do 2010 i=1,nx-1
      do 2010 k=1,nz-1
      m=nxy*(k-1)+nx*(ny-1)+i
      do 1901 nn=1,3
      do 1901 mm=1,8
      do 2099 m3=1,3
      do 2099 m8=1,8
      econ=econ+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2099  continue
1901  continue
```

```
2010  continue
c  z=nz face
      do 2021 i3=1,3
      do 2021 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
2021  continue
      do 2020 i=1,nx-1
      do 2020 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      do 1902 nn=1,3
      do 1902 mm=1,8
      do 2019 m3=1,3
      do 2019 m8=1,8
      econ=econ+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2019  continue
1902  continue
2020  continue
c  x=nx y=ny edge
      do 2031 i3=1,3
      do 2031 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.6) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      delta(i8,1)=exy*ny+exx*nx
      delta(i8,2)=eyy*ny+exy*nx
      delta(i8,3)=eyz*ny+exz*nx
      end if
2031  continue
      do 2030 k=1,nz-1
      m=nxy*k
      do 1903 nn=1,3
      do 1903 mm=1,8
      do 2029 m3=1,3
      do 2029 m8=1,8
      econ=econ+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2029  continue
1903  continue
2030  continue
c  x=nx z=nz edge
      do 2041 i3=1,3
      do 2041 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.5.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exz*nz+exx*nx
      delta(i8,2)=eyz*nz+exy*nx
      delta(i8,3)=ezz*nz+exz*nx
      end if
2041  continue
      do 2040 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      do 1904 nn=1,3
      do 1904 mm=1,8
      do 2039 m3=1,3
      do 2039 m8=1,8
      econ=econ+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2039  continue
1904  continue
2040  continue
c  y=ny z=nz edge
      do 2051 i3=1,3
      do 2051 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.3.or.i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
2051  continue
      do 2050 i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      do 1905 nn=1,3
      do 1905 mm=1,8
      do 2049 m3=1,3
      do 2049 m8=1,8
      econ=econ+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2049  continue
1905  continue
2050  continue
c  x=nx y=ny z=nz corner
      do 2061 i3=1,3
      do 2061 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.5) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      if(i8.eq.6) then
      delta(i8,1)=exx*nx+exz*nz
      delta(i8,2)=exy*nx+eyz*nz
      delta(i8,3)=exz*nx+ezz*nz
      end if
      if(i8.eq.3) then
      delta(i8,1)=exx*nx+exy*ny
      delta(i8,2)=exy*nx+eyy*ny
      delta(i8,3)=exz*nx+eyz*ny
      end if
      if(i8.eq.7) then
      delta(i8,1)=exx*nx+exy*ny+exz*nz
      delta(i8,2)=exy*nx+eyy*ny+eyz*nz
      delta(i8,3)=exz*nx+eyz*ny+ezz*nz
      end if
2061  continue
      m=nx*ny*nz
      do 1906 nn=1,3
      do 1906 mm=1,8
      do 2059 m3=1,3
      do 2059 m8=1,8
      econ=econ+0.5*delta(m8,m3)*dk(pix(m),m8,m3,mm,nn)*delta(mm,nn)
2059  continue
1906  continue
      pp(ii,jj)=econ*2.

5000  continue
      do 6000 i=1,6
      do 6000 j=i,6
      if(i.eq.j) s(i,j)=pp(i,j)
      if(i.ne.j) then
      s(i,j)=pp(i,j)-pp(i,i)-pp(j,j)
      end if
6000  continue
      do 7000 i=1,6
      do 7000 j=1,6
      pp(i,j)=0.5*(s(i,j)+s(j,i))
7000  continue
c  now map pp(i,j) into zcon(2,3,2,3)
      do 7200 i=1,2
      do 7200 j=1,2
      do 7200 mi=1,3
      do 7200 mj=1,3
      if(i.eq.1) ii=i+mi-1
      if(i.eq.2) ii=i+mi+1
      if(j.eq.1) jj=j+mj-1
      if(j.eq.2) jj=j+mj+1
      zcon(i,mi,j,mj)=pp(ii,jj)
      write(7,1) i,mi,j,mj,zcon(i,mi,j,mj)
1     format(4i5,f20.10)
7200  continue

      return
      end

c  Subroutine computes the total energy, utot, and the gradient, gb,
c  for the regular displacements as well as for the macrostrains

      subroutine energy(nx,ny,nz,ns,utot)

real u(8002,3),gb(8002,3)
real b(8002,3),T(8002,3)
real cmod(100,6,6),pk(6,8,3),eigen(100,6)
real dk(100,8,3,8,3),zcon(2,3,2,3),C
  integer*4 ib(8002,27)
  integer*2 pix(8002)

common/list3/ib
        common/list4/pix
common/list5/dk,b,C,zcon,Y
common/list6/u
common/list7/gb
        common/list8/cmod,T,eigen

        nss=ns+2

c  Do global matrix multiply via small stiffness matrices, Ah = A * h
c  The long statement below correctly brings in all the terms from
c  the global matrix A using only the small stiffness matrices.

        do 2090 m3=1,3
do 2090 m=1,nss
gb(m,m3)=0.0
2090 continue

        do 3000 j=1,3
        do 3000 n=1,3
do 3000 m=1,ns
      gb(m,j)=gb(m,j)+u(ib(m,1),n)*( dk(pix(ib(m,27)),1,j,4,n)
&+dk(pix(ib(m,7)),2,j,3,n)
&+dk(pix(ib(m,25)),5,j,8,n)+dk(pix(ib(m,15)),6,j,7,n) )+
&u(ib(m,2),n)*( dk(pix(ib(m,27)),1,j,3,n)
&+dk(pix(ib(m,25)),5,j,7,n) )+
&u(ib(m,3),n)*( dk(pix(ib(m,27)),1,j,2,n)+dk(pix(ib(m,5)),4,j,3,n)+
&dk(pix(ib(m,13)),8,j,7,n)+dk(pix(ib(m,25)),5,j,6,n) )+
&u(ib(m,4),n)*( dk(pix(ib(m,5)),4,j,2,n)
&+dk(pix(ib(m,13)),8,j,6,n) )+
&u(ib(m,5),n)*( dk(pix(ib(m,6)),3,j,2,n)+dk(pix(ib(m,5)),4,j,1,n)+
&dk(pix(ib(m,14)),7,j,6,n)+dk(pix(ib(m,13)),8,j,5,n) )+
&u(ib(m,6),n)*( dk(pix(ib(m,6)),3,j,1,n)
&+dk(pix(ib(m,14)),7,j,5,n) )+
&u(ib(m,7),n)*( dk(pix(ib(m,6)),3,j,4,n)+dk(pix(ib(m,7)),2,j,1,n)+
&dk(pix(ib(m,14)),7,j,8,n)+dk(pix(ib(m,15)),6,j,5,n) )+
&u(ib(m,8),n)*( dk(pix(ib(m,7)),2,j,4,n)
&+dk(pix(ib(m,15)),6,j,8,n) )+
&u(ib(m,9),n)*( dk(pix(ib(m,25)),5,j,4,n)
&+dk(pix(ib(m,15)),6,j,3,n) )+
&u(ib(m,10),n)*( dk(pix(ib(m,25)),5,j,3,n) )+
&u(ib(m,11),n)*( dk(pix(ib(m,13)),8,j,3,n)
&+dk(pix(ib(m,25)),5,j,2,n) )+
&u(ib(m,12),n)*( dk(pix(ib(m,13)),8,j,2,n) )+
&u(ib(m,13),n)*( dk(pix(ib(m,13)),8,j,1,n)
&+dk(pix(ib(m,14)),7,j,2,n) )+
&u(ib(m,14),n)*( dk(pix(ib(m,14)),7,j,1,n) )+
&u(ib(m,15),n)*( dk(pix(ib(m,14)),7,j,4,n)
&+dk(pix(ib(m,15)),6,j,1,n) )+
&u(ib(m,16),n)*( dk(pix(ib(m,15)),6,j,4,n) )+
&u(ib(m,17),n)*( dk(pix(ib(m,27)),1,j,8,n)
&+dk(pix(ib(m,7)),2,j,7,n) )+
&u(ib(m,18),n)*( dk(pix(ib(m,27)),1,j,7,n) )+
&u(ib(m,19),n)*( dk(pix(ib(m,27)),1,j,6,n)
&+dk(pix(ib(m,5)),4,j,7,n) )+
&u(ib(m,20),n)*( dk(pix(ib(m,5)),4,j,6,n) )+
&u(ib(m,21),n)*( dk(pix(ib(m,5)),4,j,5,n)
&+dk(pix(ib(m,6)),3,j,6,n) )+
&u(ib(m,22),n)*( dk(pix(ib(m,6)),3,j,5,n) )+
&u(ib(m,23),n)*( dk(pix(ib(m,6)),3,j,8,n)
&+dk(pix(ib(m,7)),2,j,5,n) )+
&u(ib(m,24),n)*( dk(pix(ib(m,7)),2,j,8,n) )+
&u(ib(m,25),n)*( dk(pix(ib(m,14)),7,j,3,n)
&+dk(pix(ib(m,13)),8,j,4,n)+
&dk(pix(ib(m,15)),6,j,2,n)+dk(pix(ib(m,25)),5,j,1,n) )+
&u(ib(m,26),n)*( dk(pix(ib(m,6)),3,j,7,n)
&+dk(pix(ib(m,5)),4,j,8,n)+
     &dk(pix(ib(m,27)),1,j,5,n)+dk(pix(ib(m,7)),2,j,6,n) )+
     &u(ib(m,27),n)*( dk(pix(ib(m,27)),1,j,1,n)
     &+dk(pix(ib(m,7)),2,j,2,n)+
     &dk(pix(ib(m,6)),3,j,3,n)+dk(pix(ib(m,5)),4,j,4,n)
     &+dk(pix(ib(m,25)),5,j,5,n)+
     &dk(pix(ib(m,15)),6,j,6,n)+dk(pix(ib(m,14)),7,j,7,n)+
     &dk(pix(ib(m,13)),8,j,8,n) )
3000 continue

      utot=0.0

      gtot=0.0
      g2=0.0
      g1=0.0
      do 3100 m3=1,3
      do 3100 m=1,ns
      utot=utot+0.5*u(m,m3)*gb(m,m3)+b(m,m3)*u(m,m3)
c  this is gradient of energy with respect to normal displacements
      gb(m,m3)=gb(m,m3)+b(m,m3)
3100  continue

c  compute "constant" macrostrain energy term
      C=0.0
      do 7200 i=1,2
      do 7200 j=1,2
      do 7200 mi=1,3
      do 7200 mj=1,3
      C=C+0.5*u(ns+i,mi)*zcon(i,mi,j,mj)*u(ns+j,mj)
7200  continue
      utot=utot+C
c  now add in constant term from thermal energy, Y
      utot=utot+Y
c  now add in linear term in thermal energy
      do 7171 m3=1,3
      do 7171 m=1,nss
      utot=utot+T(m,m3)*u(m,m3)
7171  continue

c  now compute gradient with respect to macrostrains
c  put in piece from first derivative of zcon quadratic term
      do 7300 i=1,2
      do 7300 mi=1,3
      sum=0.0
      do 7250 j=1,2
      do 7250 mj=1,3
      sum=sum+zcon(i,mi,j,mj)*u(ns+j,mj)
7250  continue
      gb(ns+i,mi)=sum
7300  continue
c  add in piece of gradient, for displacements as well as macrostrains,
c  that come from linear term in thermal energy
      do 3150 m3=1,3
      do 3150 m=1,nss
      gb(m,m3)=gb(m,m3)+T(m,m3)
3150  continue
c  now generate part that comes from b . u term
c  do by calling b generation with appropriate macrostrain set to 1 to
c  get that partial derivative, just use bgrad (taken from femat),
c  skip dk and zcon part
      do 8100 ii=1,6
      exx=0.0
      eyy=0.0
      ezz=0.0
      exz=0.0
      eyz=0.0
      exy=0.0
      if(ii.eq.1) exx=1.0
      if(ii.eq.2) eyy=1.0
      if(ii.eq.3) ezz=1.0
      if(ii.eq.4) exz=1.0
      if(ii.eq.5) eyz=1.0
      if(ii.eq.6) exy=1.0
      call bgrad(nx,ny,nz,ns,exx,eyy,ezz,exz,eyz,exy)
      sum=0.0
        do 8200 m3=1,3
do 8200 m=1,ns
sum=sum+u(m,m3)*b(m,m3)
8200 continue
if(ii.eq.1) gb(ns+1,1)=gb(ns+1,1)+sum
if(ii.eq.2) gb(ns+1,2)=gb(ns+1,2)+sum
if(ii.eq.3) gb(ns+1,3)=gb(ns+1,3)+sum
if(ii.eq.4) gb(ns+2,1)=gb(ns+2,1)+sum
if(ii.eq.5) gb(ns+2,2)=gb(ns+2,2)+sum
if(ii.eq.6) gb(ns+2,3)=gb(ns+2,3)+sum
8100    continue
        return
        end

c  Subroutine that carries out the conjugate gradient relaxation process
```

```
      subroutine dembx(nx,ny,nz,ns,Lstep,gg,gtest,ldemb,kkk)

      real u(8002,3),gb(8002,3),b(8002,3)
      real h(8002,3),Ah(8002,3)
      real dk(100,8,3,8,3),zcon(2,3,2,3)
      real lambda,gamma
      integer*4 ib(8002,27)
      integer*2 pix(8002)

        common/list2/h,Ah
common/list3/ib
        common/list4/pix
common/list5/dk,b,C,zcon,Y
common/list6/u
common/list7/gb

      nss=ns+2
c  Initialize the conjugate direction vector on first call to dembx only.
c  For calls to dembx after the first, we want to continue using the value
c  of h determined in the previous call.  Of course, if npoints
c  is greater than 1, then this initialization step will be run each time
c  a new microstructure is used, as kkk will be reset to 1 every time
c  the counter micro is increased.
      if(kkk.eq.1) then
      do 500 m3=1,3
      do 500 m=1,nss
      h(m,m3)=gb(m,m3)
500   continue
      end if

c  Lstep counts the number of conjugate gradient steps taken
c  in each call to dembx
      Lstep=0

      do 800 ijk=1,ldemb
      Lstep=Lstep+1

      do 290 m3=1,3
      do 290 m=1,nss
      Ah(m,m3)=0.0
290   continue
c  Do global matrix multiply via small stiffness matrices, Ah = A * h
c  The long statement below correctly brings in all the terms from
c  the global matrix A using only the small stiffness matrices.
        do 400 j=1,3
```

```
         do 400 n=1,3
do 400 m=1,ns
      Ah(m,j)=Ah(m,j)+h(ib(m,1),n)*( dk(pix(ib(m,27)),1,j,4,n)
     &+dk(pix(ib(m,7)),2,j,3,n)
     &+dk(pix(ib(m,25)),5,j,8,n)+dk(pix(ib(m,15)),6,j,7,n) )+
     &h(ib(m,2),n)*( dk(pix(ib(m,27)),1,j,3,n)
     &+dk(pix(ib(m,25)),5,j,7,n) )+
     &h(ib(m,3),n)*( dk(pix(ib(m,27)),1,j,2,n)+dk(pix(ib(m,5)),4,j,3,n)+
     &dk(pix(ib(m,13)),8,j,7,n)+dk(pix(ib(m,25)),5,j,6,n) )+
     &h(ib(m,4),n)*( dk(pix(ib(m,5)),4,j,2,n)
     &+dk(pix(ib(m,13)),8,j,6,n) )+
     &h(ib(m,5),n)*( dk(pix(ib(m,6)),3,j,2,n)+dk(pix(ib(m,5)),4,j,1,n)+
     &dk(pix(ib(m,14)),7,j,6,n)+dk(pix(ib(m,13)),8,j,5,n) )+
     &h(ib(m,6),n)*( dk(pix(ib(m,6)),3,j,1,n)
     &+dk(pix(ib(m,14)),7,j,5,n) )+
     &h(ib(m,7),n)*( dk(pix(ib(m,6)),3,j,4,n)+dk(pix(ib(m,7)),2,j,1,n)+
     &dk(pix(ib(m,14)),7,j,8,n)+dk(pix(ib(m,15)),6,j,5,n) )+
     &h(ib(m,8),n)*( dk(pix(ib(m,7)),2,j,4,n)
     &+dk(pix(ib(m,15)),6,j,8,n) )+
     &h(ib(m,9),n)*( dk(pix(ib(m,25)),5,j,4,n)
     &+dk(pix(ib(m,15)),6,j,3,n) )+
     &h(ib(m,10),n)*( dk(pix(ib(m,25)),5,j,3,n) )+
     &h(ib(m,11),n)*( dk(pix(ib(m,13)),8,j,3,n)
     &+dk(pix(ib(m,25)),5,j,2,n) )+
     &h(ib(m,12),n)*( dk(pix(ib(m,13)),8,j,2,n) )+
     &h(ib(m,13),n)*( dk(pix(ib(m,13)),8,j,1,n)
     &+dk(pix(ib(m,14)),7,j,2,n) )+
     &h(ib(m,14),n)*( dk(pix(ib(m,14)),7,j,1,n) )+
     &h(ib(m,15),n)*( dk(pix(ib(m,14)),7,j,4,n)
     &+dk(pix(ib(m,15)),6,j,1,n) )+
     &h(ib(m,16),n)*( dk(pix(ib(m,15)),6,j,4,n) )+
     &h(ib(m,17),n)*( dk(pix(ib(m,27)),1,j,8,n)
     &+dk(pix(ib(m,7)),2,j,7,n) )+
     &h(ib(m,18),n)*( dk(pix(ib(m,27)),1,j,7,n) )+
     &h(ib(m,19),n)*( dk(pix(ib(m,27)),1,j,6,n)
     &+dk(pix(ib(m,5)),4,j,7,n) )+
     &h(ib(m,20),n)*( dk(pix(ib(m,5)),4,j,6,n) )+
     &h(ib(m,21),n)*( dk(pix(ib(m,5)),4,j,5,n)
     &+dk(pix(ib(m,6)),3,j,6,n) )+
     &h(ib(m,22),n)*( dk(pix(ib(m,6)),3,j,5,n) )+
     &h(ib(m,23),n)*( dk(pix(ib(m,6)),3,j,8,n)
     &+dk(pix(ib(m,7)),2,j,5,n) )+
     &h(ib(m,24),n)*( dk(pix(ib(m,7)),2,j,8,n) )+
     &h(ib(m,25),n)*( dk(pix(ib(m,14)),7,j,3,n)
     &+dk(pix(ib(m,13)),8,j,4,n)+
     &dk(pix(ib(m,15)),6,j,2,n)+dk(pix(ib(m,25)),5,j,1,n) )+
     &h(ib(m,26),n)*( dk(pix(ib(m,6)),3,j,7,n)
     &+dk(pix(ib(m,5)),4,j,8,n)+
     &dk(pix(ib(m,27)),1,j,5,n)+dk(pix(ib(m,7)),2,j,6,n) )+
     &h(ib(m,27),n)*( dk(pix(ib(m,27)),1,j,1,n)
     &+dk(pix(ib(m,7)),2,j,2,n)+
     &dk(pix(ib(m,6)),3,j,3,n)+dk(pix(ib(m,5)),4,j,4,n)
     &+dk(pix(ib(m,25)),5,j,5,n)+
     &dk(pix(ib(m,15)),6,j,6,n)+dk(pix(ib(m,14)),7,j,7,n)+
     &dk(pix(ib(m,13)),8,j,8,n) )
400    continue

c  The above accurately gives the second derivative matrix with respect
c  to nodal displacements, but fails to give the 2nd derivative terms that
c  include the macrostrains [ du d(strain) and d(strain)d(strain) ].
c  Use repeated calls to bgrad to generate mixed 2nd derivatives terms,
c  plus use zcon in order to correct the matrix multiply and correctly bring
c  in macrostrain terms (see manual, Sec. 2.4).
      do 8100 ii=1,6
      e11=0.0
      e22=0.0
      e33=0.0
      e13=0.0
      e23=0.0
      e12=0.0
      if(ii.eq.1) e11=1.0
      if(ii.eq.2) e22=1.0
      if(ii.eq.3) e33=1.0
      if(ii.eq.4) e13=1.0
      if(ii.eq.5) e23=1.0
      if(ii.eq.6) e12=1.0
      call bgrad(nx,ny,nz,ns,e11,e22,e33,e13,e23,e12)
c  now fill in terms from matrix multiply
c  right hand sides, 1 to ns
      do 3333 m=1,ns
      do 3333 m1=1,3
      if(ii.eq.1) Ah(m,m1)=Ah(m,m1)+b(m,m1)*h(ns+1,1)
      if(ii.eq.2) Ah(m,m1)=Ah(m,m1)+b(m,m1)*h(ns+1,2)
      if(ii.eq.3) Ah(m,m1)=Ah(m,m1)+b(m,m1)*h(ns+1,3)
      if(ii.eq.4) Ah(m,m1)=Ah(m,m1)+b(m,m1)*h(nss,1)
      if(ii.eq.5) Ah(m,m1)=Ah(m,m1)+b(m,m1)*h(nss,2)
      if(ii.eq.6) Ah(m,m1)=Ah(m,m1)+b(m,m1)*h(nss,3)
3333  continue
c  now do across bottom, 1 to ns
      do 3334 m=1,ns
```

```
       if(ii.eq.1) Ah(ns+1,1)=Ah(ns+1,1)+b(m,1)*h(m,1)+
      +b(m,2)*h(m,2)+b(m,3)*h(m,3)
       if(ii.eq.2) Ah(ns+1,2)=Ah(ns+1,2)+b(m,1)*h(m,1)+
      +b(m,2)*h(m,2)+b(m,3)*h(m,3)
       if(ii.eq.3) Ah(ns+1,3)=Ah(ns+1,3)+b(m,1)*h(m,1)+
      +b(m,2)*h(m,2)+b(m,3)*h(m,3)
       if(ii.eq.4) Ah(nss,1)=Ah(nss,1)+b(m,1)*h(m,1)+
      +b(m,2)*h(m,2)+b(m,3)*h(m,3)
       if(ii.eq.5) Ah(nss,2)=Ah(nss,2)+b(m,1)*h(m,1)+
      +b(m,2)*h(m,2)+b(m,3)*h(m,3)
       if(ii.eq.6) Ah(nss,3)=Ah(nss,3)+b(m,1)*h(m,1)+
      +b(m,2)*h(m,2)+b(m,3)*h(m,3)
3334  continue
c  now do righthand corner terms, ns+1 to nss
      do 3335 m=1,2
      do 3335 m1=1,3
      if(ii.eq.1) Ah(ns+1,1)=Ah(ns+1,1)+zcon(1,1,m,m1)*h(ns+m,m1)
      if(ii.eq.2) Ah(ns+1,2)=Ah(ns+1,2)+zcon(1,2,m,m1)*h(ns+m,m1)
      if(ii.eq.3) Ah(ns+1,3)=Ah(ns+1,3)+zcon(1,3,m,m1)*h(ns+m,m1)
      if(ii.eq.4) Ah(nss,1)=Ah(nss,1)+zcon(2,1,m,m1)*h(ns+m,m1)
      if(ii.eq.5) Ah(nss,2)=Ah(nss,2)+zcon(2,2,m,m1)*h(ns+m,m1)
      if(ii.eq.6) Ah(nss,3)=Ah(nss,3)+zcon(2,3,m,m1)*h(ns+m,m1)
3335  continue

8100    continue

      hAh=0.0
      do 530 m3=1,3
      do 530 m=1,nss
      hAh=hAh+h(m,m3)*Ah(m,m3)
530   continue

      lambda=gg/hAh
      do 540 m3=1,3
      do 540 m=1,nss
      u(m,m3)=u(m,m3)-lambda*h(m,m3)
      gb(m,m3)=gb(m,m3)-lambda*Ah(m,m3)
540   continue

      gglast=gg
      gg=0.0
      do 550 m3=1,3
      do 550 m=1,nss
      gg=gg+gb(m,m3)*gb(m,m3)
550   continue
```

```
      if(gg.lt.gtest) goto 1000

      gamma=gg/gglast
      do 570 m3=1,3
      do 570 m=1,nss
      h(m,m3)=gb(m,m3)+gamma*h(m,m3)
570   continue

800   continue

1000  continue
      return
      end

c  Subroutine that computes the six average stresses and six average strains

      subroutine stress(nx,ny,nz,ns)

      real u(8002,3),uu(8,3)
      real T(8002,3),eigen(100,6)
      real dndx(8),dndy(8),dndz(8),es(6,8,3),cmod(100,6,6)
      integer*4 ib(8002,27)
      integer*2 pix(8002)

      common/list1/strxx,stryy,strzz,strxz,stryz,strxy
      common/list3/ib
      common/list4/pix
      common/list6/u
      common/list8/cmod,T,eigen
      common/list11/sxx,syy,szz,sxz,syz,sxy

      nxy=nx*ny
      nss=ns+2
      exx=u(ns+1,1)
      eyy=u(ns+1,2)
      ezz=u(ns+1,3)
      exz=u(nss,1)
      eyz=u(nss,2)
      exy=u(nss,3)

c set up single pixel strain matrix

      dndx(1)=-0.25
      dndx(2)=0.25
      dndx(3)=0.25
      dndx(4)=-0.25
      dndx(5)=-0.25
      dndx(6)=0.25
      dndx(7)=0.25
      dndx(8)=-0.25
      dndy(1)=-0.25
      dndy(2)=-0.25
      dndy(3)=0.25
      dndy(4)=0.25
      dndy(5)=-0.25
      dndy(6)=-0.25
      dndy(7)=0.25
      dndy(8)=0.25
      dndz(1)=-0.25
      dndz(2)=-0.25
      dndz(3)=-0.25
      dndz(4)=-0.25
      dndz(5)=0.25
      dndz(6)=0.25
      dndz(7)=0.25
      dndz(8)=0.25

c  Build average strain matrix, follows code in femat, but for average
c  strain over a pixel, not the strain at a point
      do 2799 n1=1,6
      do 2799 n2=1,8
      do 2799 n3=1,3
      es(n1,n2,n3)=0.0
2799  continue
      do 2797 n=1,8
      es(1,n,1)=dndx(n)
      es(2,n,2)=dndy(n)
      es(3,n,3)=dndz(n)
      es(4,n,1)=dndz(n)
      es(4,n,3)=dndx(n)
      es(5,n,2)=dndz(n)
      es(5,n,3)=dndy(n)
      es(6,n,1)=dndy(n)
      es(6,n,2)=dndx(n)
2797  continue
c  now compute average stresses and strains in each pixel
      sxx=0.0
      syy=0.0
      szz=0.0
      sxz=0.0
      syz=0.0
      sxy=0.0
      strxx=0.0
      stryy=0.0
      strzz=0.0
      strxz=0.0
      stryz=0.0
      strxy=0.0
      do 470 k=1,nz
      do 470 j=1,ny
      do 470 i=1,nx
      m=(k-1)*nxy+(j-1)*nx+i
c  load in elements of 8-vector using pd. bd. conds.
      do 9898 mm=1,3
      uu(1,mm)=u(m,mm)
      uu(2,mm)=u(ib(m,3),mm)
      uu(3,mm)=u(ib(m,2),mm)
      uu(4,mm)=u(ib(m,1),mm)
      uu(5,mm)=u(ib(m,26),mm)
      uu(6,mm)=u(ib(m,19),mm)
      uu(7,mm)=u(ib(m,18),mm)
      uu(8,mm)=u(ib(m,17),mm)
9898  continue
c  Correct for periodic boundary conditions, some displacements are wrong
c  for a pixel on a periodic boundary.  Since they come from an opposite
c  face, need to put in applied strain to correct them.
      if(i.eq.nx) then
      uu(2,1)=uu(2,1)+exx*nx
      uu(2,2)=uu(2,2)+exy*nx
      uu(2,3)=uu(2,3)+exz*nx
      uu(3,1)=uu(3,1)+exx*nx
      uu(3,2)=uu(3,2)+exy*nx
      uu(3,3)=uu(3,3)+exz*nx
      uu(6,1)=uu(6,1)+exx*nx
      uu(6,2)=uu(6,2)+exy*nx
      uu(6,3)=uu(6,3)+exz*nx
      uu(7,1)=uu(7,1)+exx*nx
      uu(7,2)=uu(7,2)+exy*nx
      uu(7,3)=uu(7,3)+exz*nx
      end if
      if(j.eq.ny) then
      uu(3,1)=uu(3,1)+exy*ny
      uu(3,2)=uu(3,2)+eyy*ny
      uu(3,3)=uu(3,3)+eyz*ny
      uu(4,1)=uu(4,1)+exy*ny
      uu(4,2)=uu(4,2)+eyy*ny
      uu(4,3)=uu(4,3)+eyz*ny
      uu(7,1)=uu(7,1)+exy*ny
      uu(7,2)=uu(7,2)+eyy*ny
      uu(7,3)=uu(7,3)+eyz*ny
      uu(8,1)=uu(8,1)+exy*ny
      uu(8,2)=uu(8,2)+eyy*ny
      uu(8,3)=uu(8,3)+eyz*ny
      end if
      if(k.eq.nz) then
      uu(5,1)=uu(5,1)+exz*nz
      uu(5,2)=uu(5,2)+eyz*nz
      uu(5,3)=uu(5,3)+ezz*nz
      uu(6,1)=uu(6,1)+exz*nz
      uu(6,2)=uu(6,2)+eyz*nz
      uu(6,3)=uu(6,3)+ezz*nz
      uu(7,1)=uu(7,1)+exz*nz
      uu(7,2)=uu(7,2)+eyz*nz
      uu(7,3)=uu(7,3)+ezz*nz
      uu(8,1)=uu(8,1)+exz*nz
      uu(8,2)=uu(8,2)+eyz*nz
      uu(8,3)=uu(8,3)+ezz*nz
      end if

c  stresses and strains in a pixel
      str11=0.0
      str22=0.0
      str33=0.0
      str13=0.0
      str23=0.0
      str12=0.0
      s11=0.0
      s22=0.0
      s33=0.0
      s13=0.0
      s23=0.0
      s12=0.0
c********compute average stress and strain tensor in each pixel*************
c  First put thermal strain-induced stresses into stress tensor
      do 465 n=1,6
      str11=str11-cmod(pix(m),1,n)*eigen(pix(m),n)
      str22=str22-cmod(pix(m),2,n)*eigen(pix(m),n)
      str33=str33-cmod(pix(m),3,n)*eigen(pix(m),n)
      str13=str13-cmod(pix(m),4,n)*eigen(pix(m),n)
      str23=str23-cmod(pix(m),5,n)*eigen(pix(m),n)
      str12=str12-cmod(pix(m),6,n)*eigen(pix(m),n)
465   continue
      do 466 n3=1,3
      do 466 n8=1,8
c  compute non-thermal strains in each pixel
      s11=s11+es(1,n8,n3)*uu(n8,n3)
      s22=s22+es(2,n8,n3)*uu(n8,n3)
      s33=s33+es(3,n8,n3)*uu(n8,n3)
      s13=s13+es(4,n8,n3)*uu(n8,n3)
      s23=s23+es(5,n8,n3)*uu(n8,n3)
      s12=s12+es(6,n8,n3)*uu(n8,n3)
      do 466 n=1,6
c  compute stresses in each pixel that include both non-thermal
c  and thermal strains
      str11=str11+cmod(pix(m),1,n)*es(n,n8,n3)*uu(n8,n3)
      str22=str22+cmod(pix(m),2,n)*es(n,n8,n3)*uu(n8,n3)
      str33=str33+cmod(pix(m),3,n)*es(n,n8,n3)*uu(n8,n3)
      str13=str13+cmod(pix(m),4,n)*es(n,n8,n3)*uu(n8,n3)
      str23=str23+cmod(pix(m),5,n)*es(n,n8,n3)*uu(n8,n3)
      str12=str12+cmod(pix(m),6,n)*es(n,n8,n3)*uu(n8,n3)
466   continue
c  Sum local stresses and strains into global stresses and strains
      strxx=strxx+str11
      stryy=stryy+str22
      strzz=strzz+str33
      strxz=strxz+str13
      stryz=stryz+str23
      strxy=strxy+str12
      sxx=sxx+s11
      syy=syy+s22
      szz=szz+s33
      sxz=sxz+s13
      syz=syz+s23
      sxy=sxy+s12
470   continue

c  Volume average global stresses and strains

strxx=strxx/float(ns)
stryy=stryy/float(ns)
strzz=strzz/float(ns)
strxz=strxz/float(ns)
stryz=stryz/float(ns)
strxy=strxy/float(ns)
sxx=sxx/float(ns)
```

```
syy=syy/float(ns)
szz=szz/float(ns)
sxz=sxz/float(ns)
syz=syz/float(ns)
sxy=sxy/float(ns)

      return
      end

c  Subroutine to count volume fractions of various phases

      subroutine assig(ns,nphase,prob)
      integer*2 pix(8002)
      real prob(100)
      common/list4/pix

do 999 i=1,nphase
prob(i)=0.0
999 continue

      do 1000 m=1,ns
      do 1000 i=1,nphase
        if(pix(m).eq.i) then
prob(i)=prob(i)+1
end if
1000    continue

do 998 i=1,nphase
prob(i)=prob(i)/float(ns)
998 continue

          return
          end

c  Subroutine to set up image of microstructure

      subroutine ppixel(nx,ny,nz,ns,nphase)
      integer*2 pix(8002)
      integer*4 ib(8002,27)
      common/list3/ib
      common/list4/pix

c  (USER)  If you want to set up a test image inside the program, instead
c  of reading it in from a file, this should be done inside this subroutine.
```

```
      nxy=nx*ny
      do 200 k=1,nz
      do 200 j=1,ny
      do 200 i=1,nx
      m=nxy*(k-1)+nx*(j-1)+i
      read(9,*) pix(m)
200   continue

c  Check for wrong phase labels--less than 1 or greater than nphase
       do 500 m=1,ns
       if(pix(m).lt.1) then
        write(7,*) 'Phase label in pix < 1--error at ',m
       end if
       if(pix(m).gt.nphase) then
        write(7,*) 'Phase label in pix > nphase--error at ',m
       end if
500    continue

      return
      end
```

9.3.4 DC3D.F

```
c  ************************  dc3d.f  ********************************
c  BACKGROUND

c  This program accepts as input a 3-d digital image, converting it
c  into a real conductor network. The conjugate gradient method
c  is used to solve this finite difference representation of Laplace's
c  equation for real conductivity problems.
c  Periodic boundary conditions are maintained.
c  In the comments below, (USER) means that this is a section of code
c  that the user might have to change for his particular problem.
c  Therefore the user is encouraged to search for this string.

c  PROBLEM AND VARIABLE DEFINITION

c  The mathematical problem that the conjugate gradient algorithm solves
c  is the minimization of the quadratic form 1/2 uAu, where
c  u is the vector of voltages, and A is generated from the bond
c  conductances between pixels. Nodes are thought of as being in the
c  center of pixels. The minimization is constrained by maintaining an
c  general direction applied electric field across the sample.
c  The vectors gx,gy,gz are bond conductances, u is the voltage array,
c  and gb,h, and Ah are auxiliary variables, used in subroutine dembx.
c  The vector pix contains the phase labels for each pixel.
c  The small vector a(i) is the volume fraction
c  of the i'th phase, and currx, curry, currz are the total volume-averaged
c  currents in the x,y, and z directions.

c  DIMENSIONS

c  The vectors gx,gy,gz,u,gb,h,Ah,list,pix are all dimensioned
c  ns2 = (nx+2)*(ny+2)*(nz+2).  This number is used, rather than the
c  system size nx x ny x nz, because an extra layer of pixels is
c  put around the system to be able to maintain periodic boundary
c  conditions (see manual, Sec. 3.3). The arrays pix and list are also
c  dimensioned this way.
c  At present the program is set up for up to 100
c  phases, but that can easily be changed by the user, by changing the
c  dimension of sigma, a, and be. Note that be has both dimensions
c  equal to each other. The parameter nphase gives the number of
c  phases being considered. The parameter ntot is the total number
c  of phases possible in the program, and should be equal to the
c  dimension of sigma, a, and be.
c  All arrays are passed to subroutines in the call statements.
```

```
c  STRONGLY RECOMMENDED:  READ MANUAL BEFORE USING THE PROGRAM!!

c  (USER)  Change these dimensions for different system sizes.  All
c  dimensions in the subroutines are passed, so do not need to be changed.
c  The dimensions of sigma, a, and be should be equal to the value of ntot.
      real gx(10648),gy(10648),u(10648),gz(10648)
      real gb(10648),h(10648),ah(10648)
      real currx,curry,currz,sigma(100,3)
      real a(100),be(100,100,3)
      integer*2 pix(10648)
      integer*4 list(10648)

c  (USER) Unit 9 is the microstructure input file, unit 7 is the
c  results output file.
      open(unit=9,file='microstructure.dat')
      open(unit=7,file='outputfile.out')

c  (USER) real image size is nx x ny x nz
      nx=20
      ny=20
      nz=20
      write(7,1111) nx,ny,nz,nx*ny*nz
1111  format(' Image is ',3i6,' No. of real sites = ',i8)
c  auxiliary variables involving the system size
      nx1=nx+1
      ny1=ny+1
      nz1=nz+1
      nx2=nx+2
      ny2=ny+2
      nz2=nz+2
      L22=nx2*ny2
c  computational image size ns2 is nx2 x ny2 x nz2
      ns2=nx2*ny2*nz2

c  (USER) set cutoff for norm squared of gradient, gtest.  gtest is
c  the stopping criterion, compared to gb*gb. When gb*gb is less
c  than gtest=abc*ns2, then the rms value of the gradient at a pixel
c  is less than sqrt(abc).
      gtest=1.0e-16*ns2

c  (USER) nphase is the number of phases being considered in the problem.
c  The values of pix(m) will run from 1 to nphase. ntot is the total
c  number of phases possible in the program, and is the dimension of
c  sigma, a, and be.
```

```
      nphase=2
      ntot=100

c  Make list of real (interior) sites, used in subroutine dembx.  The 1-d
c  labelling scheme goes over all ns2 sites, so a list of the real sites
c  is needed.
      nlist=0
      do 103 i=2,nx1
      do 102 j=2,ny1
      do 101 k=2,nz1
      m=i+(j-1)*nx2+(k-1)*L22
      nlist=nlist+1
      list(nlist)=m
101   continue
102   continue
103   continue

c  Compute average current in each pixel.
c (USER) npoints is the number of microstructures to use.

      npoints=1
      do 8000 micro=1,npoints
c Read in a microstructure in subroutine ppixel, and set up pix(m)
c  with the appropriate phase assignments.
      call ppixel(pix,nx2,ny2,nz2,a,ns2,nphase,ntot)
c  output phase volume fractions
      do 99 i=1,nphase
      write(7,299) i,a(i)
299   format(' Phase fraction of ',i3,' = ',f12.6)
99    continue

c  (USER) Set components of applied field, E = (ex,ey,ez)
      ex=1.0
      ey=1.0
      ez=1.0
      write(7,*) 'Applied field components:'
      write(7,*) 'ex = ',ex,' ey = ',ey,' ez = ',ez
c  Initialize the voltage distribution by putting on uniform field.
      do 30 k=1,nz2
      do 30 j=1,ny2
      do 30 i=1,nx2
      m=(k-1)*nx2*ny2+nx2*(j-1)+i
      u(m)=-ex*i-ey*j-ez*k
30    continue
c  (USER) input value of real conductivity tensor for each phase
```

```
c  (diagonal only). 1,2,3 = x,y,z, respectively.
      sigma(1,1)=1.0
      sigma(1,2)=1.0
      sigma(1,3)=1.0
      sigma(2,1)=0.5
      sigma(2,2)=0.5
      sigma(2,3)=0.5

c  Subroutine bond sets up conductor network in gx,gy,gz 1-d arrays
       call bond(pix,gx,gy,gz,nx2,ny2,nz2,ns2,sigma,be,nphase,ntot)

c  Subroutine dembx accepts gx,gy,gz and solves for the voltage field
c  that minimizes the dissipated energy.
      call dembx(nx2,ny2,nz2,ns2,gx,gy,gz,u,ic,gb,h,ah,list,nlist,gtest)

c  find final current after voltage solution is done
      call current(nx2,ny2,nz2,ns2,currx,curry,currz,u,gx,gy,gz)
      write(7,*) 'Average current in x direction= ',currx
      write(7,*) 'Average current in y direction= ',curry
      write(7,*) 'Average current in z direction= ',currz
      write(7,*) ic,' number of conjugate gradient cycles needed'
      call flush(7)
8000  continue
      end

c  Subroutine that performs the conjugate gradient solution routine to
c  find the correct set of nodal voltages

      subroutine dembx(nx2,ny2,nz2,ns2,gx,gy,gz,u,ic,gb,h,Ah,
     &  list,nlist,gtest)
      real gx(ns2),gy(ns2),u(ns2),gb(ns2)
      real Ah(ns2),h(ns2),gz(ns2)
      real gg,hAh,lambda,gglast,gamma,ravg,currx,curry,currz
      integer*4 list(ns2),ncheck

c  Note:  voltage gradients are maintained because in the conjugate gradient
c  relaxation algorithm, the voltage vector is only modified by adding a
c  periodic vector to it.
      L22=nx2*ny2
c  First stage, compute initial value of gradient (gb), initialize h, the
c  conjugate gradient direction, and compute norm squared of gradient vector.

      call prod(nx2,ny2,nz2,ns2,gx,gy,gz,u,gb)
      do 20 i=1,ns2
      h(i)=gb(i)
```

```
20    continue
c  Variable gg is the norm squared of the gradient vector
      gg=0.0
      do 105 k=1,nlist
      m=list(k)
      gg=gb(m)*gb(m)+gg
105   continue

c  Second stage, initialize Ah variable, compute parameter lamdba,
c  make first change in voltage array, update gradient (gb) vector

      if(gg.lt.gtest) goto 44
      call prod(nx2,ny2,nz2,ns2,gx,gy,gz,h,Ah)
      hAh=0.0
      do 205 k=1,nlist
      m=list(k)
      hAh=hAh+h(m)*Ah(m)
205   continue
      lambda=gg/hAh
      do 50 i=1,ns2
      u(i)=u(i)-lambda*h(i)
      gb(i)=gb(i)-lambda*Ah(i)
50    continue

c  third stage:  iterate conjugate gradient solution process until
c  gg < gtest criterion is satisfied.
c  (USER) The parameter ncgsteps is the total number of conjugate gradient steps
c  to go through.  Only in very unusual problems, like when the conductivity
c  of one phase is much higher than all the rest, will this many steps be
c  used.
      ncgsteps=30000

      do 33 icc=1,ncgsteps
      gglast=gg
      gg=0.0
      do 305 k=1,nlist
      m=list(k)
      gg=gb(m)*gb(m)+gg
305   continue
      call flush(7)
      if(gg.lt.gtest) goto 44
      gamma=gg/gglast
c  update conjugate gradient direction
      do 70 i=1,ns2
      h(i)=gb(i)+gamma*h(i)
```

```
70    continue
      call prod(nx2,ny2,nz2,ns2,gx,gy,gz,h,Ah)
      hAh=0.0
      do 401 k=1,nlist
      m=list(k)
      hAh=hAh+h(m)*Ah(m)
401   continue
      lambda=gg/hAh
c  update voltage, gradient vectors
      do 90 i=1,ns2
      u(i)=u(i)-lambda*h(i)
      gb(i)=gb(i)-lambda*Ah(i)
90    continue

c  (USER) This piece of code forces dembx to write out the total current and
c  the norm of the gradient squared, every ncheck conjugate gradient steps,
c  in order to see how the relaxation is proceeding. If the currents become
c  unchanging before the relaxation is done, then gtest was picked to be
c  smaller than was necessary.
      ncheck=30

      if(ncheck*(icc/ncheck).eq.icc) then
      write(7,*) icc
      write(7,*) ' gg = ',gg
c  call current subroutine
      call current(nx2,ny2,nz2,ns2,currx,curry,currz,u,gx,gy,gz)
      write(7,*) ' currx = ',currx
      write(7,*) ' curry = ',curry
      write(7,*) ' currz = ',currz
      end if
      call flush(7)
33    continue
      write(7,*) ' Iteration failed to converge after',ncgsteps,' steps'
44    continue
      ic=icc

      return
      end

c The matrix product subroutine

      subroutine prod(nx2,ny2,nz2,ns2,gx,gy,gz,xw,yw)
      real gx(ns2),gy(ns2),gz(ns2),xw(ns2),yw(ns2)

c  xw is the input vector, yw = (A)(xw) is the output vector
```

```
c  auxiliary variables involving the system size
      nx1=nx2-1
      ny1=ny2-1
      nz1=nz2-1
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      L22=nx2*ny2

c  Perform basic matrix multiplication, results in incorrect information at
c  periodic boundaries.
      do 10 i=1,ns2
      yw(i)=0.0
10    continue
      do 20 i=L22+1,ns2-L22
      yw(i)=-xw(i)*(gx(i-1)+gx(i)+gz(i-L22)+gz(i)+gy(i)+gy(i-nx2))
      yw(i)=yw(i)+gx(i-1)*xw(i-1)+gx(i)*xw(i+1)
     + +gz(i-L22)*xw(i-L22)+gz(i)*xw(i+L22)+gy(i)*xw(i+nx2)
     + +gy(i-nx2)*xw(i-nx2)
20    continue

c  Correct terms at periodic boundaries (Section 3.3 in manual)

c  x faces
      do 30 k=1,nz2
      do 30 j=1,ny2
      yw((k-1)*L22+nx2*(j-1)+nx2)=yw((k-1)*L22+nx2*(j-1)+2)
      yw((k-1)*L22+nx2*(j-1)+1)=yw((k-1)*L22+nx2*(j-1)+nx1)
30    continue

c   y faces
      do 40 k=1,nz2
      do 40 i=1,nx2
      yw((k-1)*L22+i)=yw((k-1)*L22+ny*nx2+i)
      yw((k-1)*L22+ny1*nx2+i)=yw((k-1)*L22+nx2+i)
40    continue

c  z faces
      do 50 m=1,L22
      yw(m)=yw(m+nz*L22)
      yw(m+nz1*L22)=yw(m+L22)
50    continue

      return
      end

c  Subroutine that determines the correct bond conductances that are used
c  to compute multiplication by the matrix A

      subroutine bond(pix,gx,gy,gz,nx2,ny2,nz2,ns2,sigma,be,nphase,ntot)
      real gx(ns2),gy(ns2),gz(ns2),sigma(ntot,3),be(ntot,ntot,3)
      integer*2 pix(ns2)

c  auxiliary variables involving the system size
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      nx1=nx2-1
      ny1=ny2-1
      nz1=nz2-1
      L22=nx2*ny2

c  Set values of conductor for phase(i,m)--phase(j,m) interface,
c  store in array be(i,j,m), m=1,2,3. If either phase i or j
c  has zero conductivity in the m'th direction, then be(i,j,m)= 0.0.

      do 10 m=1,3
      do 10 i=1,nphase
      do 10 j=1,nphase
      if(sigma(i,m).eq.0.0) then
      be(i,j,m)=0.0
      goto 10
      end if
      if(sigma(j,m).eq.0.0) then
      be(i,j,m)=0.0
      goto 10
      end if
      be(i,j,m)=1./(0.5/sigma(i,m)+0.5/sigma(j,m))
10    continue

c  Trim off x and y faces so that no current can flow past periodic
c  boundaries.  This step is not really necessary, as the voltages on the
c  periodic boundaries will be matched to the corresponding real voltages
c  in each conjugate gradient step.
      do 20 k=1,nz2
      do 15 j=1,ny2
      gx((k-1)*L22+nx2*(j-1)+nx2)=0.0
15    continue
      do 16 i=1,nx2
      gy((k-1)*L22+ny1*nx2+i)=0.0
16    continue
20    continue

c  Set up conductor network
c   bulk--gz
      do 30 i=1,nx2
      do 30 j=1,ny2
      do 30 k=1,nz1
      m=(k-1)*L22+(j-1)*nx2+i
      i1=i
      j1=j
      k1=k+1
      m1=(k1-1)*L22+(j1-1)*nx2+i1
      gz(m)=be(pix(m),pix(m1),3)
30    continue

c  bulk---gy
      do 40 i=1,nx2
      do 40 j=1,ny1
      do 40 k=2,nz1
      m=(k-1)*L22+(j-1)*nx2+i
      j1=j+1
      i1=i
      k1=k
      m1=(k1-1)*L22+(j1-1)*nx2+i1
      gy(m)=be(pix(m),pix(m1),2)
40    continue

c  bulk--gx
      do 50 i=1,nx1
      do 50 j=1,ny2
      do 50 k=2,nz1
      m=(k-1)*L22+(j-1)*nx2+i
      i1=i+1
      j1=j
      k1=k
      m1=(k1-1)*L22+(j1-1)*nx2+i1
      gx(m)=be(pix(m),pix(m1),1)
50    continue

      return
      end

c  Subroutine that sets up the image, either by reading it from file,
c  or generating it internally

      subroutine ppixel(pix,nx2,ny2,nz2,a,ns2,nphase,ntot)
      real a(ntot)
      integer*2 pix(ns2)

c  (USER) If you want to set up a test image inside the program, instead
c  of reading it in from a file, this should be done inside this subroutine.

c  auxiliary variables involving the system size
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      L22=nx2*ny2

c  Initialize phase fraction array.
      do 120 i=1,nphase
      a(i)=0.0
120   continue

c Use 1-d labelling scheme as shown in manual
      do 100 k=2,nz2-1
      do 100 j=2,ny2-1
      do 100 i=2,nx2-1
      m=(k-1)*L22+(j-1)*nx2+i
      read(9,*) pix(m)
      a(pix(m))=a(pix(m))+1.0
100   continue
      do 220 i=1,nphase
      a(i)=a(i)/float(nx*ny*nz)
220   continue

c  now map periodic boundaries of pix (see Section 3.3, Figure 3 in manual)
      do 110 k=1,nz2
      do 110 j=1,ny2
      do 110 i=1,nx2
      if(i.gt.1.and.i.lt.nx2) then
         if(j.gt.1.and.j.lt.ny2) then
           if(k.gt.1.and.k.lt.nz2) then
           goto 110
           end if
         end if
      end if

      k1=k
      if(k.eq.1) k1=k+nz
      if(k.eq.nz2) k1=k-nz
      j1=j
      if(j.eq.1) j1=j+ny
      if(j.eq.ny2) j1=j-ny
      i1=i
      if(i.eq.1) i1=i+nx
      if(i.eq.nx2) i1=i-nx

      m=(k-1)*L22+(j-1)*nx2+i
      m1=(k1-1)*L22+(j1-1)*nx2+i1

      pix(m)=pix(m1)
110    continue

c  Check for wrong phase labels--less than 1 or greater than nphase
       do 500 m=1,ns2
       if(pix(m).lt.1) then
        write(7,*) 'Phase label in pix < 1--error at ',m
       end if
       if(pix(m).gt.nphase) then
        write(7,*) 'Phase label in pix > nphase--error at ',m
       end if
500    continue

      return
      end

c  Subroutine to compute the total current in the x, y, and z directions

      subroutine current(nx2,ny2,nz2,ns2,currx,curry,currz,u,gx,gy,gz)
      real u(ns2),gx(ns2),gy(ns2),gz(ns2),currx,curry,currz

c  auxiliary variables involving the system size
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      L22=nx2*ny2
c  initialize the volume averaged currents
      currx=0.0
      curry=0.0
      currz=0.0

c  Only loop over real sites and bonds in order to get true total current
      do 10 k=2,nz2-1
      do 10 j=2,ny2-1
      do 10 i=2,nx2-1
      m=L22*(k-1)+nx2*(j-1)+i
c  cur1, cur2, cur3 are the currents in one pixel
      cur1=0.5*( ( u(m-1) - u(m) )*gx(m-1)+( u(m)-u(m+1) )*gx(m) )
      cur2=0.5*( ( u(m-nx2) - u(m) )*gy(m-nx2)+( u(m)-u(m+nx2) )*gy(m) )
      cur3=0.5*( ( u(m-L22) - u(m) )*gz(m-L22)+( u(m)-u(m+L22) )*gz(m) )
c  sum pixel currents into volume averages
      currx=currx+cur1
      curry=curry+cur2
      currz=currz+cur3
10    continue

      currx=currx/float(nx*ny*nz)
      curry=curry/float(nx*ny*nz)
      currz=currz/float(nx*ny*nz)

      return
      end
```

9.3.5 AC3D.F

```
c  ***********************  ac3d.f ******************************
c  BACKGROUND

c  This program accepts as input a 3-d digital image, converting it
c  into a complex conductor network. The conjugate gradient method
c  is used to solve this finite difference representation of Laplace's
c  equation for complex conductivity problems.
c  Periodic boundary conditions are maintained.
c  In the comments below, (USER) means that this is a section of code
c  that the user might have to change for his particular problem.
c  Therefore the user is encouraged to search for this string.

c  PROBLEM AND VARIABLE DEFINITION

c  The mathematical problem that the conjugate gradient algorithm solves
c  is the minimization of the quadratic form 1/2 uAu, where
c  u is the vector of voltages, and A is generated from the bond
c  conductances between pixels. Nodes are thought of as being in the
c  center of pixels. The minimization is constrained by maintaining an
c  general direction applied electric field across the sample.
c  The vectors gx,gy,gz are bond conductances, u is the voltage array,
c  and gb,h, and Ah are auxiliary variables, used in subroutine dembx.
c  The vector pix contains the phase labels for each pixel.
c  The small vector a(i) is the volume fraction
c  of the ith phase, and currx, curry, currz are the total volume-averaged
c  complex currents in the x,y, and z directions.

c  DIMENSIONS

c  The vectors gx,gy,gz,u,gb,h,Ah,list,pix are all dimensioned
c  ns2 = (nx+2)*(ny+2)*(nz+2).  This number is used, rather than the
c  system size nx x ny x nz, because an extra layer of pixels is
c  put around the system to be able to maintain periodic boundary
c  conditions (see manual, Sec. 3.3).  The arrays pix and list are also
c  dimensioned this way.  At present the program is set up for 100 phases,
c  but that can easily be changed by the user, by changing the dimensions
c  of sigma, a, and be.  Note that be has both dimensions equal to
c  each other.  The parameter nphase gives the number of phases
c  being considered.  The parameter ntot is the total number of phases
c  possible in the program, and should be equal to the dimension
c  of sigma, a, and be.  All arrays are passed to subroutines in
c  the call statements.
```

```
c  STRONGLY RECOMMENDED:  READ MANUAL BEFORE USING THE PROGRAM!!

c  (USER)  Change these dimensions for different system sizes.  All
c  dimensions in the subroutines are passed, so do not need to be
c  changed.  The dimensions of sigma, a, and be should be equal to
c  the value of ntot.
      complex gx(10648),gy(10648),u(10648),gz(10648)
      complex gb(10648),h(10648),ah(10648)
      complex currx,curry,currz,sigma(100,3),be(100,100,3)
      real a(100)
      integer*2 pix(10648)
      integer*4 list(10648)
c  (USER) Unit 9 is the microstructure input file, unit 7 is the
c  results output file.
      open(unit=9,file='microstructure.dat')
      open(unit=7,file='outputfile.out')

c  (USER) real image size is nx x ny x nz
      nx=20
      ny=20
      nz=20
c  auxiliary variables involving the system size
      nx1=nx+1
      ny1=ny+1
      nz1=nz+1
      nx2=nx+2
      ny2=ny+2
      nz2=nz+2
      L22=nx2*ny2
      write(7,1111) nx,ny,nz,nx*ny*nz
1111  format(' Image is ',3i6,' No. of real sites = ',i8)
c  computational image size ns2 is nx2 x ny2 x nz2
      ns2=nx2*ny2*nz2

c  defines the value of pi for later use
      pi=4.0*atan(1.0)

c  (USER) set cutoff for norm squared of gradient, gtest.  gtest is
c  the stopping criterion, compared to gb*gb.  When gb*gb is less
c  than gtest=abc*ns2, then the rms value of the gradient at a pixel
c  is less than sqrt(abc).
      gtest=1.0e-16*ns2

c  (USER) nphase is the number of phases being considered in the problem.
c  The values of pix(m) will run from 1 to nphase.  ntot is the total
```

```
c  number of phases possible in the program, and is the dimension of
c  sigma, a, and be.
      nphase=2
      ntot=100

c  Make list of real (interior) sites, used in subroutine dembx.  The 1-d
c  labelling scheme goes over all ns2 sites, so a list of the real sites
c  is needed.
      nlist=0
      do 103 i=2,nx1
      do 102 j=2,ny1
      do 101 k=2,nz1
      m=i+(j-1)*nx2+(k-1)*L22
      nlist=nlist+1
      list(nlist)=m
101   continue
102   continue
103   continue

c  Compute average current in each pixel.
c (USER) npoints is the number of microstructures to use.
c  nfreq is the number of frequencies to be computed.
c  The program is set up assuming that the effective
c  conductivity is going to be solved for at several different
c  frequencies on the same microstructure.

      npoints=1
      do 400 micro=1,npoints
c  Read in a microstructure in subroutine ppixel, and set up
c  pix(m) with the appropriate phase assignments.
      call ppixel(pix,nx2,ny2,nz2,a,ns2,nphase,ntot)
c  output phase volume fractions
      do 99 i=1,nphase
      write(7,299) i,a(i)
299   format(' Phase fraction of ',i3,' = ',f12.6)
99    continue

c  (USER) Set components of applied field, E = (ex,ey,ez)
      ex=1.0
      ey=1.0
      ez=1.0
      write(7,*) 'Applied field components:'
      write(7,*) 'ex = ',ex,' ey = ',ey,' ez = ',ez
c  Initialize the voltage distribution by putting on uniform field.
c  Only do this for the first frequency considered, thereafter use the
```

```
c  previous frequency's voltages as a starting point.
      do 30 k=1,nz2
      do 30 j=1,ny2
      do 30 i=1,nx2
      m=(k-1)*nx2*ny2+nx2*(j-1)+i
      u(m)=-ex*i-ey*j-ez*k
30    continue

c  (USER) Set how many frequencies need to be computed.
      nfreq=50
c  Loop over desired frequencies.
      do 300 nf=1,nfreq
c  (USER) Define frequency to use each time.  Alter this statement to get
c  different frequencies. The frequencies are in Hz, according to
c  the units used for sigma.
      w=10.**((nf-1)*11.4/49.-3.)
c  convert to angular frequency
      w=w*2.*pi
      write(7,*) 'No.',nf, ' angular frequency = ',w,' radians'
c  (USER) input value of complex conductivity tensor for each phase
c  (diagonal only). 1,2,3 = x,y,z,  respectively.
      sigma(1,1)=cmplx(1.0,10.*w)
      sigma(1,2)=cmplx(1.0,10.*w)
      sigma(1,3)=cmplx(1.0,10.*w)
      sigma(2,1)=cmplx(0.5,1.*w)
      sigma(2,2)=cmplx(0.5,1.*w)
      sigma(2,3)=cmplx(0.5,1.*w)

c  bond() sets up conductor network in gx,gy,gz 1-d arrays
       call bond(pix,gx,gy,gz,nx2,ny2,nz2,ns2,sigma,be,nphase,ntot)

c  Subroutine dembx accepts gx,gy,gz and solves for the voltage field
c  that minimizes the dissipated energy.  As a starting point for u,
c  the voltage vector, each frequency uses the voltages left over from the
c  previous minimization.  This can often reduce total run time dramatically,
c  compared to starting with a new voltage vector each time, as in do loop
c  30 above.
      call dembx(nx2,ny2,nz2,ns2,gx,gy,gz,u,ic,gb,h,ah,list,nlist,gtest)

c  find final current after voltage solution is done
      call current(nx2,ny2,nz2,ns2,currx,curry,currz,u,gx,gy,gz)
      write(7,*) 'Average current in x direction= ',currx
      write(7,*) 'Average current in y direction= ',curry
      write(7,*) 'Average current in z direction= ',currz
      write(7,*) ic,' number of conjugate gradient cycles needed'
      call flush(7)
300   continue
400   continue
      end

c  Subroutine that performs the conjugate gradient solution routine to
c  find the correct set of nodal voltages

      subroutine dembx(nx2,ny2,nz2,ns2,gx,gy,gz,u,ic,gb,h,Ah,
     &  list,nlist,gtest)
      complex gx(ns2),gy(ns2),u(ns2),gb(ns2)
      complex Ah(ns2),h(ns2),gz(ns2)
      complex gg,hAh,lambda,gglast,gamma,ravg,currx,curry,currz
      integer*4 list(ns2),ncheck

c  Note:  voltage gradients are maintained because in the conjugate gradient
c  relaxation algorithm, the voltage vector is only modified by adding a
c  periodic vector to it.
      L22=nx2*ny2
c  First stage, compute initial value of gradient (gb), initialize h, the
c  conjugate gradient direction, and compute norm squared of gradient vector.

      call prod(nx2,ny2,nz2,ns2,gx,gy,gz,u,gb)
      do 20 i=1,ns2
      h(i)=gb(i)
20    continue
c  Complex variable gg is the norm squared of the gradient vector
      gg=cmplx(0.0,0.0)
      do 105 k=1,nlist
      m=list(k)
      gg=gb(m)*gb(m)+gg
105   continue

c  Second stage, initialize Ah variable, compute parameter lamdba,
c  make first change in voltage array, update gradient (gb) vector

      if(abs(real(gg)).lt.gtest) goto 44
      call prod(nx2,ny2,nz2,ns2,gx,gy,gz,h,Ah)
      hAh=cmplx(0.0,0.0)
      do 205 k=1,nlist
      m=list(k)
      hAh=hAh+h(m)*Ah(m)
205   continue
      lambda=gg/hAh
      do 50 i=1,ns2
```

```
        u(i)=u(i)-lambda*h(i)
        gb(i)=gb(i)-lambda*Ah(i)
50      continue

c  third stage:  iterate conjugate gradient solution process until
c  real(gg) < gtest criterion is satisfied.
c  (USER) The parameter ncgsteps is the total number of conjugate gradient steps
c  to go through.  Only in very unusual problems, like when the conductivity
c  of one phase is much higher than all the rest, will this many steps be
c  used.
        ncgsteps=30000

        do 33 icc=1,ncgsteps
        gglast=gg
        gg=cmplx(0.0,0.0)
        do 305 k=1,nlist
        m=list(k)
        gg=gb(m)*gb(m)+gg
305     continue
        call flush(7)
        if(abs(real(gg)).lt.gtest) goto 44
        gamma=gg/gglast
c  update conjugate gradient direction
        do 70 i=1,ns2
        h(i)=gb(i)+gamma*h(i)
70      continue
        call prod(nx2,ny2,nz2,ns2,gx,gy,gz,h,Ah)
        hAh=cmplx(0.0,0.0)
        do 401 k=1,nlist
        m=list(k)
        hAh=hAh+h(m)*Ah(m)
401     continue
        lambda=gg/hAh
c  update voltage, gradient vectors
        do 90 i=1,ns2
        u(i)=u(i)-lambda*h(i)
        gb(i)=gb(i)-lambda*Ah(i)
90      continue
c  (USER) This piece of code forces dembx to write out the total current and
c  the norm of the gradient squared, every ncheck conjugate gradient steps,
c  in order to see how the relaxation is proceeding. If the currents become
c  unchanging before the relaxation is done, then gtest was picked to be
c  smaller than was necessary.
        ncheck=30
```

```
      if(ncheck*(icc/ncheck).eq.icc) then
      write(7,*) icc
      write(7,*) ' gg = ',gg
c  call current subroutine
      call current(nx2,ny2,nz2,ns2,currx,curry,currz,u,gx,gy,gz)
      write(7,*) ' currx = ',currx
      write(7,*) ' curry = ',curry
      write(7,*) ' currz = ',currz
      end if
      call flush(7)
33    continue
      write(7,*) ' Iteration failed to converge after',ncgsteps,' steps'
44    continue
      ic=icc

      return
      end

c The matrix product subroutine

      subroutine prod(nx2,ny2,nz2,ns2,gx,gy,gz,xw,yw)
      complex gx(ns2),gy(ns2),xw(ns2),gz(ns2),yw(ns2)

c  xw is the input vector, yw = (A)(xw) is the output vector

c  auxiliary variables involving the system size
      nx1=nx2-1
      ny1=ny2-1
      nz1=nz2-1
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      L22=nx2*ny2

c  Perform basic matrix multiplication, results in incorrect information at
c  periodic boundaries.
      do 10 i=1,ns2
      yw(i)=cmplx(0.0,0.0)
10    continue
      do 20 i=L22+1,ns2-L22
      yw(i)=-xw(i)*(gx(i-1)+gx(i)+gz(i-L22)+gz(i)+gy(i)+gy(i-nx2))
      yw(i)=yw(i)+gx(i-1)*xw(i-1)+gx(i)*xw(i+1)
     + +gz(i-L22)*xw(i-L22)+gz(i)*xw(i+L22)+gy(i)*xw(i+nx2)
     + +gy(i-nx2)*xw(i-nx2)
20    continue
```

```
c  Correct terms at periodic boundaries (Section 3.3 in manual)

c  x faces
      do 30 k=1,nz2
      do 30 j=1,ny2
      yw((k-1)*L22+nx2*(j-1)+nx2)=yw((k-1)*L22+nx2*(j-1)+2)
      yw((k-1)*L22+nx2*(j-1)+1)=yw((k-1)*L22+nx2*(j-1)+nx1)
30    continue

c   y faces
      do 40 k=1,nz2
      do 40 i=1,nx2
      yw((k-1)*L22+i)=yw((k-1)*L22+ny*nx2+i)
      yw((k-1)*L22+ny1*nx2+i)=yw((k-1)*L22+nx2+i)
40    continue

c  z faces
      do 50 m=1,L22
      yw(m)=yw(m+nz*L22)
      yw(m+nz1*L22)=yw(m+L22)
50    continue

      return
      end

c  Subroutine that determines the correct bond conductances that are used
c  to compute multiplication by the matrix A

      subroutine bond(pix,gx,gy,gz,nx2,ny2,nz2,ns2,sigma,be,nphase,ntot)
      complex gx(ns2),gy(ns2),gz(ns2),sigma(ntot,3),be(ntot,ntot,3)
      integer*2 pix(ns2)

c  auxiliary variables involving the system size
      L22=nx2*ny2
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      nx1=nx2-1
      ny1=ny2-1
      nz1=nz2-1

c  Set values of conductor for phase(i)--phase(j) interface,
c  store in array be(i,j,3). If either phase i or j has zero conductivity,
c  then be(i,j,3)=0.
```

```
      do 10 m=1,3
      do 10 i=1,nphase
      do 10 j=1,nphase
      if(real(sigma(i,m)).eq.0.0.and.aimag(sigma(i,m)).eq.0.0) then
      be(i,j,m)=cmplx(0.0,0.0)
      goto 10
      end if
      if(real(sigma(j,m)).eq.0.0.and.aimag(sigma(j,m)).eq.0.0) then
      be(i,j,m)=cmplx(0.0,0.0)
      goto 10
      end if
      be(i,j,m)=1./(0.5/sigma(i,m)+0.5/sigma(j,m))
10    continue

c  Trim off x and y faces so that no current can flow past periodic
c  boundaries.  This step is not really necessary, as the voltages on the
c  periodic boundaries will be matched to the corresponding real voltages
c  in each conjugate gradient step.
      do 20 k=1,nz2
      do 15 j=1,ny2
      gx((k-1)*L22+nx2*(j-1)+nx2)=cmplx(0.0,0.0)
15    continue
      do 16 i=1,nx2
      gy((k-1)*L22+ny1*nx2+i)=cmplx(0.0,0.0)
16    continue
20    continue

c  Set up conductor network
c   bulk--gz
      do 30 i=1,nx2
      do 30 j=1,ny2
      do 30 k=1,nz1
      m=(k-1)*L22+(j-1)*nx2+i
      i1=i
      j1=j
      k1=k+1
      m1=(k1-1)*L22+(j1-1)*nx2+i1
      gz(m)=be(pix(m),pix(m1),3)
30    continue

c  bulk---gy
      do 40 i=1,nx2
      do 40 j=1,ny1
      do 40 k=2,nz1
      m=(k-1)*L22+(j-1)*nx2+i
      j1=j+1
      i1=i
      k1=k
      m1=(k1-1)*L22+(j1-1)*nx2+i1
      gy(m)=be(pix(m),pix(m1),2)
40    continue

c  bulk--gx
      do 50 i=1,nx1
      do 50 j=1,ny2
      do 50 k=2,nz1
      m=(k-1)*L22+(j-1)*nx2+i
      i1=i+1
      j1=j
      k1=k
      m1=(k1-1)*L22+(j1-1)*nx2+i1
      gx(m)=be(pix(m),pix(m1),1)
50    continue

      return
      end

c  Subroutine that sets up the image, either by reading it from file,
c  or generating it internally

      subroutine ppixel(pix,nx2,ny2,nz2,a,ns2,nphase,ntot)
      real a(ntot)
      integer*2 pix(ns2)

c  (USER) If you want to set up a test image inside the program, instead
c  of reading it in from a file, this should be done inside this subroutine.

c  auxiliary variables involving the system size
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      L22=nx2*ny2
c  Initialize phase fraction array.
      do 120 i=1,nphase
      a(i)=0.0
120   continue
c Use 1-d labelling scheme as shown in manual
      do 100 k=2,nz2-1
      do 100 j=2,ny2-1
      do 100 i=2,nx2-1
      m=(k-1)*L22+(j-1)*nx2+i
      read(9,*) pix(m)
      a(pix(m))=a(pix(m))+1.0
100   continue
      do 220 i=1,nphase
      a(i)=a(i)/float(nx*ny*nz)
220   continue

c  now map periodic boundaries of pix (see Section 3.3, Figure 3 in manual)
      do 110 k=1,nz2
      do 110 j=1,ny2
      do 110 i=1,nx2
      if(i.gt.1.and.i.lt.nx2) then
         if(j.gt.1.and.j.lt.ny2) then
           if(k.gt.1.and.k.lt.nz2) then
           goto 110
           end if
         end if
      end if

      k1=k
      if(k.eq.1) k1=k+nz
      if(k.eq.nz2) k1=k-nz
      j1=j
      if(j.eq.1) j1=j+ny
      if(j.eq.ny2) j1=j-ny
      i1=i
      if(i.eq.1) i1=i+nx
      if(i.eq.nx2) i1=i-nx

      m=(k-1)*L22+(j-1)*nx2+i
      m1=(k1-1)*L22+(j1-1)*nx2+i1

      pix(m)=pix(m1)
110    continue

c  Check for wrong phase labels--less than 1 or greater than nphase
       do 500 m=1,ns2
       if(pix(m).lt.1) then
        write(7,*) 'Phase label in pix < 1--error at ',m
       end if
       if(pix(m).gt.nphase) then
        write(7,*) 'Phase label in pix > nphase--error at ',m
       end if
500    continue
```

```
      return
      end

c  Subroutine to compute the total current in the x, y, and z directions

      subroutine current(nx2,ny2,nz2,ns2,currx,curry,currz,u,gx,gy,gz)
      complex u(ns2),gx(ns2),gy(ns2),gz(ns2),currx,curry,currz
      complex cur1,cur2,cur3

c  auxiliary variables involving the system size
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      L22=nx2*ny2
c  initialize the volume averaged currents
      currx=cmplx(0.0,0.0)
      curry=cmplx(0.0,0.0)
      currz=cmplx(0.0,0.0)

c  Only loop over real sites and bonds in order to get true total current
      do 10 k=2,nz2-1
      do 10 j=2,ny2-1
      do 10 i=2,nx2-1
      m=L22*(k-1)+nx2*(j-1)+i
c  cur1, cur2, and cur3 are the currents in one pixel
      cur1=0.5*( ( u(m-1)-u(m) )*gx(m-1)+( u(m)-u(m+1) )*gx(m) )
      cur2=0.5*( ( u(m-nx2)-u(m) )*gy(m-nx2)+( u(m)-u(m+nx2) )*gy(m) )
      cur3=0.5*( ( u(m-L22)-u(m) )*gz(m-L22)+( u(m)-u(m+L22) )*gz(m) )
c  sum pixel currents into volume averages
      currx=currx+cur1
      curry=curry+cur2
      currz=currz+cur3
10    continue

      currx=currx/float(nx*ny*nz)
      curry=curry/float(nx*ny*nz)
      currz=currz/float(nx*ny*nz)

      return
      end
```

9.3.6 GAUSS.F

```
c***********************   gauss.f  **********************************

      real x(4000),w(4000),xi(4000),wi(4000)
      open(unit=62,file='gauss20.dat')
      data pi/3.14159 26535 89793 23846 26434/

c*********************************************************************
c   This program (gauss.f) evaluates zeros and weights of Legendre     *
c   polynomials to be used in Gaussian integrals.                      *
c   Pick N to be the order of the Gaussian quadrature that you         *
c   wish to use.  File output will contain Gaussian points in the      *
c   first column, weights in the second column.                        *
c   The limit on N is currently 4000.  If a larger value of N is       *
c   desired (not recommended), then change the dimensions of           *
c   x,w,xi, and wi.                                                    *
c   This program was supplied by Professor W.W. Repko, of the          *
c   Michigan State University Physics and Astronomy Department (1983). *
c*********************************************************************

c  Number N of Gaussian points desired.
      N=20
      M=(N+1)/2

c  Subroutine grule calculates weights and points

      call grule(N,x,w)

      do 10 i=1,M
      xi(i)=-x(i)
      xi(i+M)=x(M+1-i)
      wi(i)=w(i)
      wi(i+M)=w(M+1-i)
10    continue

c  output weights and points

      xc=0.0
      do 20 i=1,n
      write(62,3) xi(i),wi(i)
3     format(' ',2f20.12)

c  test with integral from -1 to 1 of exp(x), should be e - 1/e = 2.3504024
```

```
      xc=xc+wi(i)*exp(xi(i))
20    continue
      print *,' Numerical value of integral of exp(x) from -1 to 1 = '
      print *,xc
      print *,' Actual value is 2.3504024'

      end
      subroutine grule(N,x,w)
      real x(N),w(N)
      data pi/3.14159 26535 89793 23846 26434/
      data eps/1.e-14/
      dn=float(N)
      M=(N+1)/2
      e1=dn*(dn+1.0)
      do 10 i=1,M
      t=(4.0*float(i)-1.0)*pi/(4.0*dn+2.0)
      x0=(1.0-(1.0-1.0/dn)/(8.0*dn*dn))*cos(t)
20    call legendr(N,x0,pn,pnm1,pnp1)
      den=1.0-x0*x0
      d1=dn*(pnm1-x0*pn)
      dpn=d1/den
      d2pn=(2.0*x0*dpn-e1*pn)/den
      u=pn/dpn
      v=d2pn/dpn
      x1=x0-u*(1.0+0.50*u*v)
      if((x1-x0).lt.eps) go to 30
      x0=x1
      go to 20
30    x0=x1
      call legendr(N,x0,pn,pnm1,pnp1)
      x(i)=x0
10    w(i)=2.0*(1.0-x0**2)/(dn*pnm1)**2
      if(M+M.gt.N) x(M)=0.0
      return
      end
      subroutine legendr(N,x,pn,pnm1,pnp1)
      pkm1=1.0
      pk=x
      do 10 k=2,N
      t1=x*pk
      pkp1=t1-pkm1-(t1-pkm1)/float(k)+t1
      pkm1=pk
10    pk=pkp1
      pn=pk
      pnm1=pkm1
```

```
t1=x*pn
pnp1=t1-pnm1-(t1-pnm1)/float(k+1)+t1
return
end
```

9.3.7 BURN3D.F

```
c*************************  burn3d.f  *****************************

c  PROBLEM DEFINITION

c  In a random multi-phase structure, a question that is important
c  is whether a particular phase percolates through the microstructure
c  or not.  This program is designed to answer that question, for
c  a general 3-D multi-phase random microstructure.  The burning
c  algorithm checks whether a percolation threshold exists in a periodic
c  image. The program searches all three directions, using periodic
c  boundary conditions in the two perpendicular directions for
c  each burn.

c  Variables

c  There are a maximum of "ntot" phases possible, numbered 1,2,3,...
c  The label of the phase being burned is "phase", and is an input variable.
c  The value assigned to burned pixels is "burned" and is equal to ntot+1.
c  To burn on more than one phase at a time, just use
c  "phase2", "phase3", etc., and check for these values too, whenever
c  the value "phase" is checked for. (See manual)
c  A value of 0 is assigned to the variables percx, percy, and
c  percz for non-continuity, and 1 for percolation (continuity) in the
c  given direction.

c  Dimensions

c  (USER) The variable pix is dimensioned the size of the system.
c  The variables old and new are dimensioned 1/10 the size of the system,
c  but with three components. (A minimum of 1000 should be used
c  to dimension these variables, so that small systems will have enough
c  computation room.) These array dimensions should be changed
c  simultaneously in all subroutines using a global replacement statement.

c  (USER) Dimensions of main arrays
        integer*2 pix(1000000),old(100000,3),new(100000,3)
        integer*2 burned,phase

c  (USER) Unit = 9 is the input file, unit 7 is the output file
        open(unit=9,file='microstructure.dat')
        open(unit=7,file='output.out')

c  (USER) system size ns = nx x ny x nz
```

```
      nx=100
      ny=100
      nz=100
      ns=nx*ny*nz

c  (USER) Identify the label of the phase to be burned
      phase=1
c  (USER) Total number of phases allowed in problem
      ntot=100
c  Label of burned pixel
      burned=ntot+1

c  Read in microstructure file
      do 330 k=1,nz
      do 330 j=1,ny
      do 330 i=1,nx
      m=nx*ny*(k-1)+nx*(j-1)+i
      read(9,*) pix(m)
330   continue

c  Call the subroutine that actually does the burning
      call fire(pix,nx,ny,nz,percz,percy,percx,new,old,phase,burned)
c  Output the values of perc*, showing the continuity of the three
c  principal directions
      write(7,*) ' percx = ',percx
      write(7,*) ' percy = ',percy
      write(7,*) ' percz = ',percz
      end

c  This subroutine does the actual burning.  The burning starts with the
c  k=1 or j=1 or i=1 plane, and then iteratively burnes through the system,
c  until there are no more acessible pixels to be burned.  To be burned,
c  a pixel must be next to a previously burned pixel.

      subroutine fire(pix,nx,ny,nz,percz,percy,percx,new,old,
     &  phase,burned)

      integer*2 pix(1000000)
      integer*2 old(100000,3),new(100000,3)
      integer*2 in(6),jn(6),kn(6),phase,burned

c  System size
      ns=nx*ny*nz

c  Direction labels to check for burning path (nearest neighbor information
c  in 3-D digital system).
       in(1)=-1
       in(2)=1
       in(3)=0
       in(4)=0
       in(5)=0
       in(6)=0
       jn(1)=0
       jn(2)=0
       jn(3)=-1
       jn(4)=1
       jn(5)=0
       jn(6)=0
       kn(1)=0
       kn(2)=0
       kn(3)=0
       kn(4)=0
       kn(5)=-1
       kn(6)=1

c  Initialize percolation flags
       percx=0.0
       percy=0.0
       percz=0.0

       do 3000 ijk=1,3
c  Build up first burned pixels from i or j or k=1 layer, according to
c  choice of ijk (ijk = 1, k = 1;  ijk = 2, j = 1;  ijk = 3, i = 1).
c  Store(i,j,k) labels in array old().

       iold=0

       if(ijk.eq.1) then
       n2=ny
       n1=nx
       end if
       if(ijk.eq.2) then
       n2=nz
       n1=nx
       end if
       if(ijk.eq.3) then
       n2=ny
       n1=nz
       end if
```

```
      do 1000 jj=1,n2
      do 1000 ii=1,n1
      if(ijk.eq.1) then
      i=ii
      j=jj
      k=1
      end if
      if(ijk.eq.2) then
      i=ii
      j=1
      k=jj
      end if
      if(ijk.eq.3) then
      i=1
      j=jj
      k=ii
      end if
      m=nx*ny*(k-1)+nx*(j-1)+i

      if(pix(m).eq.phase) then
      pix(m)=burned
      iold=iold+1
      old(iold,1)=i
      old(iold,2)=j
      old(iold,3)=k
      end if
1000  continue

c  If no pixels burned in first layer, then phase can't possibly percolate,
c  so move to next direction
      if(iold.eq.0) goto 3000

c  Now start building up new burned pixels from old set of burned pixels,
c  thus propagating the fire.
60    inew=0
      do 100 i=1,iold
      ii=old(i,1)
      jj=old(i,2)
      kk=old(i,3)
c  check all six nearest neighbors of previously burned pixel
      do 90 n=1,6
      i1=ii+in(n)
      j1=jj+jn(n)
      k1=kk+kn(n)
```

```
c  Periodic boundary conditions
c  (USER)  Can replace with hard boundary conditions
c  if desired to remove periodicity. Keep the ijk if statements and the
c  goto 90 statements, and change the other if statements to have
c  goto 90 as well. That way the program does not allow "wrappping"
c  around to find a neighbor. (See manual)
      if(ijk.eq.1) then
      if(k1.lt.1.or.k1.gt.nz) goto 90
      if(i1.lt.1) i1=i1+nx
      if(i1.gt.nx) i1=i1-nx
      if(j1.lt.1) j1=j1+ny
      if(j1.gt.ny) j1=j1-ny
      end if
      if(ijk.eq.2) then
      if(j1.lt.1.or.j1.gt.ny) goto 90
      if(i1.lt.1) i1=i1+nx
      if(i1.gt.nx) i1=i1-nx
      if(k1.lt.1) k1=k1+nz
      if(k1.gt.nz) k1=k1-nz
      end if
      if(ijk.eq.3) then
      if(i1.lt.1.or.i1.gt.nx) goto 90
      if(k1.lt.1) k1=k1+nz
      if(k1.gt.nz) k1=k1-nz
      if(j1.lt.1) j1=j1+ny
      if(j1.gt.ny) j1=j1-ny
      end if

c  Store (i,j,k) labels of newly burned pixels in array new().
      m1=nx*ny*(k1-1)+nx*(j1-1)+i1
      if(pix(m1).eq.phase) then
      pix(m1)=burned
      inew=inew+1
      new(inew,1)=i1
      new(inew,2)=j1
      new(inew,3)=k1
      end if
90    continue
100   continue

c  If new pixels were burned, then transfer labels to old() array, start
c  burning process over again.
      if(inew.gt.0) then
      iold=inew
      do 150 i=1,inew
      old(i,1)=new(i,1)
      old(i,2)=new(i,2)
      old(i,3)=new(i,3)
150   continue
      goto 60
      end if

c  If no new burned pixels, then check to see if the last layer of the image
c  has any burned pixels in it.  If so, then there is continuity.  If not,
c  then there is no continuity.

      if(ijk.eq.1) then
      n2=ny
      n1=nx
      end if
      if(ijk.eq.2) then
      n2=nz
      n1=nx
      end if
      if(ijk.eq.3) then
      n2=ny
      n1=nz
      end if

      do 30 jj=1,n2
      do 30 ii=1,n1
      if(ijk.eq.1) then
      i=ii
      j=jj
      k=nz
      end if
      if(ijk.eq.2) then
      i=ii
      j=ny
      k=jj
      end if
      if(ijk.eq.3) then
      i=nx
      j=jj
      k=ii
      end if
      m=nx*ny*(k-1)+nx*(j-1)+i
      if(pix(m).eq.burned) then
      if(ijk.eq.1) percz=1.0
      if(ijk.eq.2) percy=1.0
      if(ijk.eq.3) percx=1.0
      end if
30    continue

c  Restore burned pixels back to their original label
      call restore(pix,ns,phase,burned)

3000  continue
      return
      end

c This subroutine restores the burned pixels back to their original, unburned
c value (phase).

      subroutine restore(pix,ns,phase,burned)
      integer*2 pix(1000000),phase,burned

      do 10 m=1,ns
      if(pix(m).eq.burned) pix(m)=phase
10    continue
      return
      end
```