HMD for Computer Modeling

Alfred Paul Steffens-Jr

Table of Contents

Chapter 1

Downloading and Installing HMD

§1. HMD Homepage.

HMD is freely available under the GNU public license. You may download the source code and user's manual from the HMD homepage at

http://www.heldeneng.com

Only the source code is available. The source code package is a standard GNU autotoolset package, which means that you can easily build the software on your Unix-like operating system with the standard commands

```
./configure
make
make install
```

If you want to run HMD on Microsoft Windows, then you will need to install the Cygwin unix emulator. I have used it. It installs like a charm and works great. After you have Cygwin on you windows machine you can build HMD with the standard build commands listed above just as if you were running Unix.

§2. Required Libraries.

GNU Readline

HMD requires the READLINE package for the interactive console prompt. This package should be installed on most Linux distributions, but I don't know for sure which distributions have it. I know for sure that Slackware comes with READLINE already installed. If your distribution doesn't have it, then you need to download it from the GNU site and build/install it on your machine. Or if you don't want READLINE you can build HMD without it but will not have an interactive prompt.

LAPACK

This is a well-known linear algebra package freely available at http://www.netlib.org/lapack

It was written and debugged by mathematicians E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenny, D. Sorensen. LAPACK is used in Matlab, as well as other matrix algebra software programs such as Scilab and Octave and ALGAE. HMD requires the LAPACK (Linear Algebra) library. You must go to the LAPACK webpage and download the file lapack.tgz. It is free. But you must build it on your computer before you can build HMD. So, download

http://www.netlib.org/lapack/lapack.tgz

into the directory of your choice, say, /home/LAPACK. Now perform the following steps:

1. gzip -d lapack.tgz

2. tar xvf lapack.tar

The following build commands assume that you are using Linux. If you are using some other flavor of Unix (or Cygwin on MS Windows) then consult the LAPACK documentation. At present I haven't ported HMD to any other platform (although this should all work on Cygwin).

3. cp make.inc make.inc.orig
4. cp install/make.inc.LINUX make.inc
5. make

Note: the make command is supposed to build and test everything. In my experience there will be a minor problem with the BLAS library not being built. If this happens you can run the make command again in the BLAS subdirectory. Check that you now have the two library files

```
lapack_LINUX.a     blas_LINUX.a
```

If the file blas_LINUX.a is missing, then change directory to ./BLAS/SRC and run

```
make
```

again. Now you can move these two library files into a library directory of your choosing, let's say LAPACK-LIBS. You will need to remember this directory path and the names of the two libraries when you build HMD.

GMSH

HMD is a model solver. So where are the neat wire-frame 3D images I wanted to see using a computer modeling tool? HMD uses another program for this. It's called GMSH. It is available from www.geuz.org/gmsh and can be downloaded for free. When used with HMD, GMSH is used as a standalone program, not a library, so you don't have to worry about building it. Just download it and install it.

You create a mesh file with GMSH. This file will have a file extension .msh. This file is what HMD needs to complete its finite element model. The GMSH mesh file name is specified as an HMD script command.

GMSH is needed only when using the finite element solver. If you will be using the cellular grid solver only then you will not need GMSH.

Chapter 2

HMD as a Programmable Workspace

§1. Starting the program.

HMD can be used as a batch processor or interactively. Batch mode is simply a matter of entering the script file name at the command line:

```
hmd -f scriptfile
```

To run HMD interactively, just use the **-i** argument:

```
hmd -i
```

When you run HMD interactively you can still process script files with the *source* command. For example, to execute the scriptfile mentioned above we would use

```
> source scriptfile ;
```

To exit the program, type exit.

§2. Interactive Command Prompt.

In interactive mode, HMD operates with a command line interface similar to Scilab and other software that uses a workspace. The workspace has the capability to process commands, functions and variables. For eample,

```
> x = 20;
> y = 10;
> z = x + y;
> msgprint z;
```

You must use a semicolon at the end of the statement. In some interactive languages like Matlab and Scilab, if you leave off the semicolon the answer is automatically printed to the console. In HMD you must use the *msgprint* command to see the result of a computation. All commands in HMD obey a simple syntax. The command name is first followed by a list of comma-separared parameters:

> **command_name** *paramater_1, paramater_2, ...*

For example, you could use the *msgprint* command in the following way

```
> msgprint  'x + y = ',  z;
```

which would print x + y = 30 to the screen.

The interactive workspace environment gives you some convenience while working between commands. Its pupose is mainly for storing and displaying variables and doing miscellaneous calculations. The operations that can be performed in interactive mode are just a subset of those available in script processing mode.

§3. Writing HMD Scripts.

An HMD script is a list of source code to be executed. The HMD script language has many familiar features to C programmers. You can declare variables, define functions, perform math operations, and perform indirection. But these capabilities are meant only to support the main purpose of HMD, which is to solve differential equations. The programmability features of HMD were not introduced so that you can write whole applications in HMD as if it were another coding language like Fortran or C. With this in mind you may be willing to forgive the HMD developers for not provided the full syntax support of the C language or the full matrix programmability of Scilab. Here is an example of an HMD script that does not solve any differential equations, bit shows some of its programmability features.

```
      //——————————————————————————————————————————
      // A Simple Programming Example
      //——————————————————————————————————————————

      //
      // Declare variables
      //
1     double x;
      double y;
      double z;

      //
      // Initialize values
      //
2     x = 2.0;
      y = 3.0;

      //
      // Define a function
      //
3     function multiply_this { double a1, double a2 } {
4         double answer;

5         answer = a1 * a2;

6         return answer;
7     }

      //
      // do processing
      //
8     if ( x == 2.0 ) {

9         z = multiply_this( x, y );
10            msgprint z;
       }

       //----------------------------------------------
       // End
       //----------------------------------------------
```

The first thing to note are the obvious differences from what you would expect from the C language:

Funtions

You use curly brackets instead of parenthesis for defining functions (but not calling them). You define a function with the keyword *function* but without any type specifier. The first bracket that starts the function definition must be on the same line with the function keyword (similar to tcl/tk). The closing bracket for the function definition or conditional must be the first character on a line by itself.

Variable Declarations

At line 1 of the listing we see that variables were declared just like in the C language. The variable type that can be declared include the following list

```
uchar
char
ushort
short
uint
int
ulong
long
float
double
complex
doublecomplex
```

However, if you don't need special types and just want to do floating point math, then you can declare variables on the fly (like in Scilab) which default to type *double.*

```
x = 2.0;
```

will automatically declare x in the workspace and make it a type *double.*

```
y = < 2.0 %i 7.0 >;
```

will automatically declare y in the workspace and make it a type *doublecomplex.*

Redirection

The HMD interpreter uses redirection similar to the C language. For conditionals you have **if**, **else if**, and **else**. There is no switch statement. For looping you have **while**. There is no for statement or do-while statement. The difference between the C language syntax and the syntax in HMD is that you must use curly brackets to enclose the conditional code. In C, you can execute one following statement without using curly brackets. But in HMD you must always use the curly brackets. Also, the first curly bracket must be on the same line as the redirection identifier (I am not trying to enforce a particular coding standard, it was just easier to write the parser this way).

Chapter 3

Solving Differential Equations

§1. The Equations of Mathematical Physics.

HMD was created especially for the purpose of solving a particular class of differential equations. These have been referred to as *the equations of mathematical physics* because they arise so frequenty in the study of physics. These differential equations are usually partial differential equations of the second order and are the fundamental equtions in such disciplines as quantum mechanics, fluid mechanics, the mechanics of continuum solids, acoustics, electromagnetic fields, and thermodynamics (heat transfer). Mathematicians have classified these equations broadly into three types according to their time derivatives.

Elliptic

$$\frac{\partial^2\psi}{\partial x^2}+\frac{\partial^2\psi}{\partial y^2}+\frac{\partial^2\psi}{\partial z^2}=f \tag{1}$$

Parabolic

$$\frac{\partial^2\psi}{\partial x^2}+\frac{\partial^2\psi}{\partial y^2}+\frac{\partial^2\psi}{\partial z^2}=f+\alpha\frac{\partial\psi}{\partial t} \tag{2}$$

Hyperbolic

$$\frac{\partial^2\psi}{\partial x^2}+\frac{\partial^2\psi}{\partial y^2}+\frac{\partial^2\psi}{\partial z^2}=f+\alpha\frac{\partial^2\psi}{\partial t^2} \tag{3}$$

The sum of second-order partial derivatives is usually written in abbreviated form with the Laplacian operator (in cartesian coordinates)

$$\nabla^2\equiv\frac{\partial^2\psi}{\partial x^2}+\frac{\partial^2\psi}{\partial y^2}+\frac{\partial^2\psi}{\partial z^2}$$

where

$$\nabla\equiv\vec{i}\frac{\partial}{\partial x}+\vec{j}\frac{\partial}{\partial y}+\vec{k}\frac{\partial}{\partial z}$$

with which the following examples of partial differential equations in mathematical physics may be written (see "Mathematical Methods for Physicists", 4th edition, Arfken and Weber, Academic Press, 1995)

Laplaces's Equation

$$\nabla^2 \psi = 0 \tag{4}$$

This equation occurs in studies of

a. electrostatics, magnetostatics
b. hydrodynamics (irrotational flow of perfect fluid and surface waves)
c. heat flow
d. gravitation

Poisson's Equation

$$\nabla^2 \psi = -f \tag{5}$$

This is the nonhomogeneous analog of Laplace's equation, being used where there are sources present.

Helmholtz Equation

$$\nabla^2 \psi \pm k^2 = 0 \tag{6}$$

This equation can be found from the wave equation or diffusion equation by transforming its time dependence into the fourier domain. This occurs in

a. elastic waves in solids including vibrating strings, bars, membranes
b. sound or acoustics
c. electromagnetic waves
d. nuclear reactors

Diffusion Equation

$$\nabla^2 \psi = \frac{1}{a^2} \frac{\partial \psi}{\partial t} \tag{7}$$

This equation is found in problems of heat transfer and gas dynamics.

Wave Equation

$$\nabla^2 \psi = \frac{1}{a^2} \frac{\partial^2 \psi}{\partial t^2} \tag{8}$$

This equation is found in problems of wave propagation.

Nonhomogeneous Wave Equation

$$\nabla^2 \psi = \frac{1}{a^2} \frac{\partial^2 \psi}{\partial t^2} - f \tag{9}$$

This equation is found in problems of the radiation of electromagnetic waves, sound waves, etc.

The Schrödinger Equation

$$-\frac{\hbar^2}{2m}\nabla^2\psi + V\psi = i\hbar\frac{\partial\psi}{\partial t} \tag{10}$$

which is, of course, the fundamental equation used in the quantum mechanics of atoms and elementary particles.

§2. Transformation to a Scalar Potential.

It should be noted that the function ψ is a scalar and is, generally speaking, a *potential field.* When we encounter differential equations involving vectors, for example, the electric field or a fluid velocity field, we *define* a scalar potential such that

$$\vec{F} = -\nabla\psi \tag{10}$$

from which the *divergence* operation encountered in the analysis of the vector field is transformed into a second-order partial derivative of the scalar potential

$$\nabla\cdot\vec{F} = -\nabla^2\psi \tag{11}$$

The use of a potential field representation is advantageous due to the well-developed theory that exists. The essential features of the potential field is that it is continuous and its first-order derivative is continuous except where there are sources. In ordinary language, this means that if you subdivide a region into smaller regions, the potential values at the boundary of any subregion must match the boundary values of its nearest neighbors. If a source exists in one of the subregions, then the first-order derivitive of the potential changes at that point (the value of the source tells you by how much the derivative changes).

In addition to the continuity property, potential fields are *local* (a term from physics implying that action induced by distant objects cannot happen). The value of the potential within a differentially small subregion is the (weighted) average of the values at its boundaries, except if there are sources present. If there is a source withing this subregion, then we must add a (weighted) contribution from it.

§3. Numerical Representation of the Second-Order PDE.

In the HMD solver the partial differential equations discussed above are represented by a general function equation of the form

$$k(\vec{x})\nabla^2\psi(\vec{x}) = F(t;\vec{x},\frac{\partial\psi}{\partial\vec{x}}) \tag{12}$$

This is a compact way of saying that the only thing that is really differenced is the Laplacian operator and everything else in the equation is grouped on the right-hand side as part of $F(t;\vec{x},\frac{\partial\psi}{\partial\vec{x}})$. The multiplier $k(\vec{x})$ is the weight assigned to the gradient at a particular point. In engineering this is often called a stiffness; in electrostatics it corresponds to the dielectric constant or permitivity.

In equation 12 above the nonhomogeneous term, F, is shown as a function of the first derivatives of the potential,

$$\frac{\partial\psi}{\partial\vec{x}} \equiv \frac{\partial\psi}{\partial x_1}, \frac{\partial\psi}{\partial x_2}, \dots \frac{\partial\psi}{\partial x_n}$$

The HMD solver is not designed for including the time derivatives in the function F. If the equation is a time-dependent wave equation then the equation can be transformed into fourier space so that the time derivative is replaced by a term $-\omega^2\psi$. The resulting Helmholtz equation would then be solved for normal modes and frequencies. You should think of the HMD solver as capable of solving *spacial* problems. To time step each spacial solution as with an ODE-type solver is not within the capabilities of the HMD solver. Or you could experiment with treating time as a spatial dimension.

The core algorithm of the HMD solver suggests a grid composed of cells. Equation (12) is solved locally at each cell and then its boundary values are matched to its nearest neighbors. Each cell has its own nodal Green's function solution that is solved based on its current (trial) boundary values. For linear problems the Green's function need only be calculated once. The solution is then comprised of the two-step interation of updating the boundary values from the values of the nearest neighbors, then recalculating the local solution.

Recall from your undergraduate theoretical physics the Green's function "magic rule" solution of the Poisson equation,

$$\psi(\vec{x}) = -\int_s \psi(\vec{x}')\frac{\partial G(\vec{x},\vec{x}')}{\partial n'}d^2x' + \int_v f(\vec{x}')G(\vec{x},\vec{x}')d^3x'. \tag{13}$$

Those readers who are familiar with the green's function solution of partial differential equations will note that equation (13) is valid only for Dirichtlet boundary conditions. For those readers who have never seen the green's function solution, the function $G(\vec{x},\vec{x}')$ is the *Green's function.* It is a propagator that produces the effect of an impulse source at location $\vec{x}'$ at the observation point at $\vec{x}$. If there are no sources, then only the surface integral is needed. If there are sources, then both integrals are needed. Note that the surface integral is *always* required. Infinte boundary conditions as used in the ordinary finite element method will need to be discussed later.

Now consider that we integrate Poisson's equation in a infinitesimal volume. We first represent the equation as a difference. To get an understanding of the derivation, first look at the one-dimensional case. Imagine just three points with a value of the potential, ψ, assigned at each point, $\psi_1 = \psi(x_1), \psi_2 = \psi(x_2), \psi_3 = \psi(x_3)$. Between each point we may imagine an *element* of length δx with a *weight* of $k(x)$. The numerical representation can be written as follows.

$$\Big[\frac{k_b(\psi_3-\psi)}{\delta x} - \frac{k_a(\psi-\psi_1)}{\delta x}\Big]\Big[\frac{1}{\delta x'}\Big] = f(x_2) \tag{14}$$

In equation (14) we denote the potential at x_2 as $\psi_2 \equiv \psi$ because we wish to solve the potential at the point $x = x_2$. The forcing function $f(x)$ is evaluated only at $x = x_2$. The construction here implies that we will solve for the value ψ at x_2 given that we know the values of ψ at x_1 and x_3 and the value of $f(x_2)$. Integrating equation (14) over $\delta x'$ results in the following equation.

$$\frac{k_b(\psi_3-\psi)}{\delta x} - \frac{k_a(\psi-\psi_1)}{\delta x} = f(x_2)\delta x' \tag{15}$$

Integrating equation (15) over δx results in

$$k_b(\psi_3 - \psi) - k_a(\psi - \psi_1) = f(x_2)\delta x \delta x' \tag{16}$$

and now solving for the potential ψ yields the local Green's function solution.

$$\psi = \frac{k_b}{k_a + k_b}\psi_3 + \frac{k_a}{k_a + k_b}\psi_1 - f\frac{\delta x}{k_a + k_b}\delta x' \tag{17}$$

Equation (17) tells us that the value of the potential at x_2 is the weighted sum of the values of its nearest neighbors and adjusted for the effect of a change in slope at x_2.

In order to better see the resemblence to the Green's function magic rule we can show the three-dimensional equivalent of equation (17). If we write the numerical Laplacian with u_1, ψ, u_3 along x, v_1, ψ, v_3 along y, and w_1, ψ, w_3 along z, we have

$$\frac{\frac{k_b(u_3 - \psi)}{\delta x} - \frac{k_a(\psi - u_1)}{\delta x}}{\delta x'} + \frac{\frac{q_b(v_3 - \psi)}{\delta y} - \frac{q_a(\psi - v_1)}{\delta y}}{\delta y'} + \frac{\frac{r_b(w_3 - \psi)}{\delta z} - \frac{r_a(\psi - w_1)}{\delta z}}{\delta z'} = f \tag{18}$$

Now integrate twice over the infinitesimal volume as above and solve for ψ. The solution is

$$\begin{aligned}\psi = (k_a u_1 + k_b u_3)\Big(\frac{\delta y'\delta z'}{H}\Big)\delta y \delta z + (q_a v_1 + q_b v_3)\Big(\frac{\delta x'\delta z'}{H}\Big)\delta x \delta z \\ +(r_a w_1 + r_b w_3)\Big(\frac{\delta x'\delta y'}{H}\Big)\delta x \delta y - f\Big(\frac{\delta x'\delta y'\delta z'}{H}\Big)\delta x \delta y \delta z\end{aligned} \tag{19}$$

where

$$H = (k_a + k_b)\delta y'\delta z'\delta y \delta z + (q_a + q_b)\delta x'\delta z'\delta x \delta z + (r_a + r_b)\delta x'\delta y'\delta x \delta z. \tag{20}$$

Now compare equation (19) to equation (13). In the HMD algorithm, the factor $\delta x'\delta y'\delta z'/H$ is defined as the numerical Green's function valid locally at a node point. The HMD algorithm calculates equation (19) at each node (refered to as a cell) and is defined as a *deduce* step. It will be seen that this calculation of ψ at a node depends on the node values of the nearest neighbors. After a deduce operation is performed on all the nodes the *local* boundary values, i.e., k_a, k_b, q_a, q_b, r_a, r_b, must be updated before the next iteration. The operation of updating the local boundary values with the node values of the nearest neighbors is defined as a *link* step in the HMD algorithm. Therefore, one complete iteration is a link step followed by a deduce step. In other words, these two steps are repeated over and over until the desired accuracy in the model is achieved.

Chapter 4

The Cell Grid Solver

§1. Introduction.

The cell grid solver is the primary solver algorithm in HMD. I began writing HMD as a finite element solver (meaning the kind of finite element techniques familiar to structural engineers but with continuum elements), but the limitations of the finite element technique soon became apparent. The cell grid solver works under the premise of a grid as an array of cells. The idea of cellular automata in a rectangular array (inspired by Stephen Wolfram's book, A New Kind of Science) suggested cellular automata as a way of solving potential theory problems. The cell rule is simply to match its boundary values with those of its nearest neighbors. There is nothing surprisingly new about this, nor is this really a cellular automata algorithm.

The cell grid is different from the usual concept of a grid, but for the most part the user of this software need not be alarmed. There are a few properties to remember when using the grid. A *cell* is primarily a point in space, with its own potential value and its own Green's function. It has connection points, two in each dimension, that connect it to its neighbors. The connection points themselves are objects that provide influence functions for the cell "nucleus", if you will, where the influence function may be used for a boundary value (boundary value regarded as applying to the local cell only) or used as a weighted influence function in the usual sense.

A single cell is first defined and given properties that will be inherited by the cells it "grows" later to form a grid. The first cell is defined with at least two steps. First the equation to be solved is defined; second, the boundary conditions are set. The cell is then grown to fill an N-dimensional cube. Then the array of cells, now forming a grid, is solved by repeatedly matching boundary values (link) and recalculating the potential value (deduce). The iteration is stopped at an accuracy of the user's choosing. The values of the grid may then be copied into an array and saved to a file.

§2. Defining a Cell.

To define a cell takes two steps. The first step is to define the equation that will be solved. Recall equation (12) in chapter 3.

$$k(\vec{x})\nabla^2\psi(\vec{x}) = F(t;\vec{x},\frac{\partial\psi}{\partial\vec{x}})$$

The cell grid algorithm understands this basic equation only. The first step in defining a cell is a command that incorporates the information in the above equation with the exception of the weight, $k(\vec{x})$. The weight information will be entered in the second step of the definition. Write the equation without the weight.

$$\nabla^2\psi(\vec{x}) = F(t;\vec{x},\frac{\partial\psi}{\partial\vec{x}}) \tag{1}$$

Now we enter a command that gives the cells name, ψ ("psi"), the Laplacian operator, the number of dimensions, and the function F.

```
psi   Y:   <dimension>   ∧   F
```

In the above pseudocode specification, Y: is equvalent to the Laplacian operator. Think of the wedge as like the equal sign. We enclose the word "dimension" in the <> symbols to designate that is is optional. If you leave out the dimension specifier, then the dimension defaults to 1.

As an example that would be used in an HMD script, consider the one-dimensional Laplace's equation.

$$\frac{\partial^2 \psi}{\partial x^2} = 0 \tag{2}$$

The cell definition command for this would be the following code line.

```
psi   :Y   1   ∧   0 ;
```

If we use instead the two-dimensional Laplace's equation,

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = 0 \tag{3}$$

then we would use the following HMD command.

```
psi   :Y   2   ∧   0 ;
```

Or, consider a three-dimensional Poisson's equation with a constant nonhomogeneous term.

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} + \frac{\partial^2 \psi}{\partial z^2} = 7 \tag{4}$$

The HMD command line for this case would be the following code line.

```
psi   :Y   3   ∧   7 ;
```

A Poisson's equation with a general nonhomogeneous term that is a function of x, y, and z, becomes more involved. Now we must define a function in the HMD workspace that will later be called by the HMD solver. The Poisson's equation of the form

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} + \frac{\partial^2 \psi}{\partial z^2} = F(x, y, x) \tag{5}$$

will be specified in the HMD script with the following line.

```
psi   :Y   3   ∧   My_Function, 3, $1, $2, $3 ;
```

In the above command, "My_Function" is the name of user-defined function. The number 3 following the function name specifies the number of arguments that follow. The first argument, $1, means the x coordinate of the cell. Similarly, the second argument, $2, means the y coordinate of the cell, and $3 means the cell's z coordinate. It will be seen that the the user-defined function, "My_Function", is a *callback* function. The user writes this function to take three input arguments. The function assumes that it has been passed the cell's x, y, and z coordinates and returns its calculated value. The HMD solver algorithm will call this function at each cell location. It knows what arguments that need to be passed to the function because these were specified in the above cell definition.

It will be seen that the above code line is the general method for entering an equation resembling equations (1) and (2). The callback function may take additional location-dependent arguments. To specify that the cell's grid spacing be passed to the function, one would use &1, &2, etc, instead of $1, $2, etc. To pass the cell's own value to the callback function (eigenvalue problem), one would write the argument as $0_name ($0_psi in the above example). Or to pass the first derivative (difference) of the cell's value in x, y, or z, one would give the argument as &1_psi, or &2_psi, or &3_psi.

§3. Setting the Boundary Conditions.

The boundary conditions for the cell should be set immediately after the cell definition, before any other operations are attempted on the cell (such as growing the cell or starting the solution). The cell's boundary conditions as well as its weights are set with an *anchor* command. An anchor command is comprised of the cell's name followed by the anchor symbol, _| (an underscore followed by a vertical line), followed by a list of parameters that specify the boundary conditions and weights for each dimension as follows.

psi   _| bc_{x1}, wt_{x1}, bc_{x2}, wt_{x2}, bc_{y1}, wt_{y1}, . . . ;

Think of the list of parameters as pairs of negative and positive terms for each dimension. For example, in dimension 1 which is the x dimension, the negative and positive pairs correspond to left and right of the cell. In dimension 2, which is the y dimension, the negative and positive pairs correspond to above and below the cell. But the weights are also specified on the anchor line so there are 4, not 2 parameters for each dimension. For example, for a one-dimensional cell with boundaries set to zero the following anchor command would be used.

```
psi   _|   0, 1, 0, 1 ;
```

The values of zero in the above command are setting the left and right boundaries to zero. The ones in the above command are setting the weights on either side of the cell to one. Since the weight values are multiplied with the boundary values, the above command corresponds to no weights. This would also coorespond to a material with a stiffness (or permitivity) of unity.

When any one of the boundary value or weight value entries is required to be a function, then the single entry becomes several entries that include the function name followed by the number of function arguments, followed by the arguments themselves. For example, in the above simple, one- dimensional example let us replace the zero boundary condition for the x dimension on the left with a function that takes two arguments.

```
psi _| My_Func, 2, myarg1, myarg2, 1, 0, 1 ;
```

Comparing the above anchor command with the previous example we see that where there was a single zero specifying the left boundary value, now there are 4 entries: My_Func, 2, myarg1, and myarg2. Then the remaining anchor entries are as before. One might reflect that that is why the number of function arguments must be specified: so that the parser knows where the logical argument ends and the next logical argument begins.

Suppose now that we want the same function to provide the boundary values for both boundaries. Then the anchor command would look like the following line.

```
psi _| My_Func, 2, myarg1, myarg2, 1, My_Func, 2, myarg1, myarg2, 1 ;
```

We see in the above example that, again, what we previously had given as a single entry is now 4 entries. Remember that there are still just four logical arguments, but that the 4 entries My_Func, 2, myarg1, and myarg2 form a single argument.

These boundary conditions are specified at the beginning of the setup procedure on a single cell. After the cell is grown into a grid the boundary conditions will be applied only to the actual boundaries of the model. As the grid is grown from a single cell each new cell that is reproduced carries with it the boundary value information, but a boundary value is set only if the cell happens to be on the actual boundary.

Weight specifications, on the other hand, are applied everywhere in the model. In the above example, if the values of one are replaced by some other constant, or variables, or function specifications, then these will be the weights applied throughout the model.

The boundary values and the weights are calculated only once during the first iteration, by default. This behavior can be changed so that either the boundary values or the weights or both are calculated on every iteration with the *initcell* command.

§4. Growing a Cell Grid.

A cell grid is *grown* from a single cell one layer at a time. The grow command has a special operator in the HMD solver. The grow operator is an arrow pointing to the right.

$$->$$

For example, with a cell named ψ, the grow command would look the following line.

$$psi->$$

To grow multiple layers we would either repeat this command on subsequent lines, or we could put it in a while loop.

```
i = 0;
```

```
while  (i < 10)  {
        psi  − > ;
        i = i +1 ;
}
```

Note that the grow operator's component symbols, the dash followed by the > sign must not contain any spaces between them.

§5. Solving the Cell Grid.

Now the cell (i.e., the seed of the grid) has been defined, and the boundary conditions set. The seed cell has been grown into a grid. The process of solving the cell grid will cause the effect of the boundaries and the sources to propagate throughout the grid. The grid gradually converges, that is, relaxes to its equilibrium. This solution process requires iteration of two basic steps: link and deduce.

The link step has the effect of updating each cell member of the grid. Each cell has member objects that contain the values of its local boundaries. In a link step the local boundary objects of a particular cell are updated with the cell values of its nearest neighbors. For example, a one-dimensional cell will have two boundary objects: one on the left and one on the right (negative x and positive x). During a link step this cell's left boundary object will be updated with the neighboring cell value to the left; the cell's right-side boundary object will be updated with the neighboring cell value on the right. The link step has the purpose of completely isolating each cell when its potential value is *deduced.* The link step is merely an alternative approach to maintaining a separate grid that would contain the previous or "last" values that become inputs to the "next" iteration.

The *link* step has its own operator defined in the HMD solver: a greater-than sign and a less-than sign, ><. A script command to link the cells in a grid whose seed cell was names *psi* would look like the following line.

```
>< psi ;
```

Note that the link operator comes before the cell name, not after it.

The second and final step in a solution iteration is a deduce step. The deduce step calculates an individual cell's potential value based on the values stored in its local boundary objects and the local value of the forcing fucntion. The deduce step uses these values with the local Green's function to calculate the cell's potential. The operator used for executing a deduce step is two question marks, ??. The command code for executing a deduce step on the cell grid called *psi* is shown in the following line.

```
?? psi ;
```

Now to solve the cell grid names *psi* we must iterate the two steps, link and deduce, repeatedly until we achieved the desired accuracy. This is shown in the following example.

```
i = 0;
while  (i < 100)  {
        >< psi ;
        ?? psi ;
```

```
          i = i +1 ;
}
```

In the above loop we are just guessing that 100 iterations would be sufficient for convergence. The HMD solver has a built-in function for evaluating the accuracy of the solution. The function *cellaccuracy* scans the group of cells and returns the least value of accuracy found in the group. This accuracy is calculated as the absolute value of the difference between the cell's value and its previous value divided by the previous value. The above solution iteration loop can be re-written with the *cellaccuracy* function as follows.

```
double err;
err = 0;
while  (err > 0.001)  {
          >< psi ;
          ?? psi ;
          err = cellaccuracy(psi) ;
}
```

§6. Manipulating the Cell Grid Solution Data.

After the cell grid has been solved, the data (potential) contained at each cell can be extracted from the cell group and placed into an array. This is done by extracting either a line or a plane. A line can be copied into a one-dimensional array, and a plane can be copied into a two-dimensional array. The command that obtains a line from the cell group is called *getcell_line*. The command that obtains a plane from the cell group is called *getcell_plane*. In either case the command will create this new array for you and copy the data into it. You may then save this data into an ascii data file for manipulation with other software tools.

The *getcell_line* command is used with the following syntax.

```
getcell_line < cell_name > , < array_name > , < coord_flag >, < dim_number >,
     < location_1 >, < location_2 >, < i >, < j >, < k >, . . . ;
```

The arguments to the above *getcell_line* command follow the following syntax:

< cell_name >	the name of the cell (cell group)
< array_name >	the name of the array to be created
< coord_flag >	value of 1 means create a coordinate array also
< dim_number >	which coordinate axis the line is parallel to
< location_1 >	the start start index of the line
< location_2 >	the end index of the line
< i >, < j >, < k >, . . .	a point on the line

It should be noted that the argument for < dim_number > requires the numerical equivalent of "x-axis", "y-axis", etc, where $x = 1$, $y = 2$, $z = 3$, etc. The line to be extracted must be parallel to one of the coordinate axes. The argument for < location_1 > is the location index of the cell where the line is to start.

Each cell has identifying indices ($i = x$, $j = y$, $k = z$, etc) for each dimension. Similarly, the argument for < location_2 > is the location index of the cell where the line is to end. The last arguments comprise a list of cell indices for a single cell somewhere on the line (you choose it). These indices uniquely identify the location of the line to be extracted.

For example, consider a one-dimensional model from which you would like to extract the data. The *getcell_line* command in this case has only one line to copy from but which segment of the line is still to be determined by your input parameters. Consider that the following model is a line in the x dimension whose cells have indices numbered from -10 to +10. To extract the whole line into an array you would use the following command.

```
int     ixdim, i1, i2;
ixdim = 1;
i1 = -10;
i2 = 10;

getcell_line psi , 'vpsi' , 1, ixdim, i1, i2, 0;
```

We see in the above example that the name of the cell group is *psi*. The name of the array to be created will be called *vpsi*. The literal integer value of 1 that follows will cause a second array to be created containing the coordinates (not the location indices) belonging to the cells. This is useful when plotting the data. The second array will automatically have a name taken from the name of the primary array but with an _s appended. In this case the second array will be named *vpsi_s*. The next argument, *ixdim*, has a value of one and specifies that the x dimension is used. There is actually no other choice: the dimension number of a one-dimensional model is automatically dimension number 1. The next two arguments, *i1* and *i2*, specify the endpoints of the segment to be extracted. Finally, the 0 value of the last argument specifies the cell with location index 0 as the cell that uniquely identifies this line. In one dimension this last argument is, of course, redundant. However, if you were to specify a location index in this last argument that is not within the *i1* and *i2* endpoints you will get an error.

For example, consider a three-dimensional model from which you would like to extract a line for plotting. This will be a line parallel to the z axis that intersects the x, y plane at the location indices $2, 3$ in that plane.

```
int     izdim, k1, k2;
izdim = 3;
k1 = -10;
k2 = 10;

getcell_line psi , 'vpsi' , 1, izdim, k1, k2, 2, 3, 0;
```

In the above example we can see that the dimension number is now 3, specifying a line parallel to the z axis. The line endpoints are the same but now pertain to the z direction. There are now 3 indices, $2, 3, 0$, specifying a cell that uniquely identifies the line. This cell is located in the x, y plane and identifies the line that passes through the point $2, 3$. The position *along* the line is given as $z = 0$ but could have been any other index between -10 and $+10$.

To extract a plane of data from a model the model must, of course, have at least two dimensions. The *getcell_plane* command has very similar syntax to the *getcell_line* command but with extra parameters for specifying the second dimension. Consider a two-dimensional model from which we would like to extract the data.

```
int      ixdim, iydim, i1, i2, j1, j2;
ixdim = 1;
iydim = 2;
i1 = -10;
i2 = 10;
j1 = -10;
j2 = 10;

getcell_plane psi , 'vpsi' , 0, ixdim, iydim, i1, i2, j1, j2, 0, 0;
```

As with the *getcell_line* command shown above, we see that the first arguments must be the name of the cell group, *psi*, followed by the name to be given to the array, *vpsi*. In this example we will extract only the cell potentials, not the cell coordinates also, so that the next argument is given as 0. The following two arguments, *ixdim* and *iydim* specify the x, y plane. These arguments are redundant in this case because we are extracting a plane from a model that is itself only a plane. The next two arguments, *i1* and *i2* give the range in x, and the next two arguments, *j1* and *j2* give the y range. Finally, the last two arguments, 0 and 0, specify the location indices of a cell that uniquely identifies the plane to be extracted. But in this case these two arguments are redundant since there is only one plane in the model.

After the data has been extracted from a cell group, you can save the data to a disk file. The *save* command is used for saving one-dimensional arrays as columns in an ascii file. For example, the two one-dimension arrays *vpsi* and *vpsi_s* can be saved as a paired data set to a disk file called "vpsi.dat" with the following command.

```
save 'vpsi.dat', vpsi_s, vpsi;
```

The *save2* command will save a two-dimension array (matrix) to a disk file. The following example shows how to save the matrix called *vpsi* to a disk file called "vpsi.dat".

```
save2 'vpsi.dat', vpsi;
```

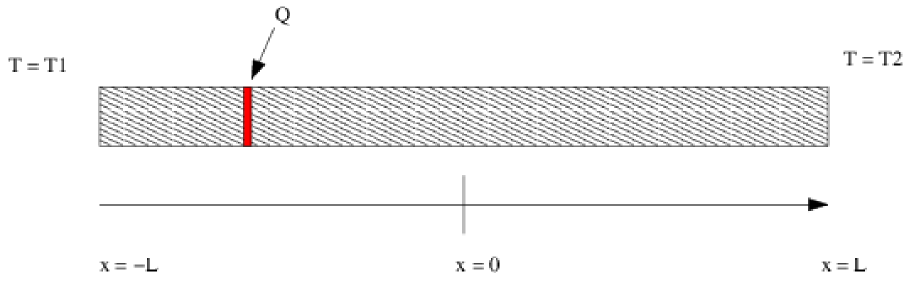
§7. 1-D Poisson's Equation - Heat Transfer.

Consider Poisson's equation for one dimension:

$$\frac{\partial^2 \psi}{\partial x^2} = -f. \tag{6}$$

The presence of the minus sign on the nonhomogeneous term is annoying but insures that the "potential change" is in the same direction as the "force". One should convince one's self of the previous statement by integrating equation (6) once over x and see that a positive force $f \delta x'$ acting at position x' produces a negative change in slope at x', which for zero Dirichtlet boundary conditions (the ends are pinned), must result in a positive value of the potential at x'.

To illustrate the one-dimensional Poisson's equation we will examine the steady state heat transfer through a thin bar. The bar is so thin that there is no variation in the y and z dimensions. The sides of the bar are insulated so well that no heat is conducted through them, except at a single, small region where a heat source is placed. The ends of the bar can conduct heat and are held at temperatures T_1 and T_2. Let the thermal conductivity, k, be uniform throughout the bar.



Temperature distribution through a thin bar

The differential equation to be solved for this problem is

$$k\frac{\partial^2 T}{\partial x^2} = -Q. \tag{7}$$

where T is the temperature and Q is the heat source at x'. First we should solve the problem with the formulas of mathematical physics so that we can make a comparison with the numerical answer.

Problem 4-1. Show that the Green's function for equation (7) is

$$x\frac{L-x'}{2kL} + \frac{L-x'}{2k} \qquad x < x'$$

$$-x\frac{L+x'}{2kL} + \frac{L+x'}{2k} \qquad x > x'$$

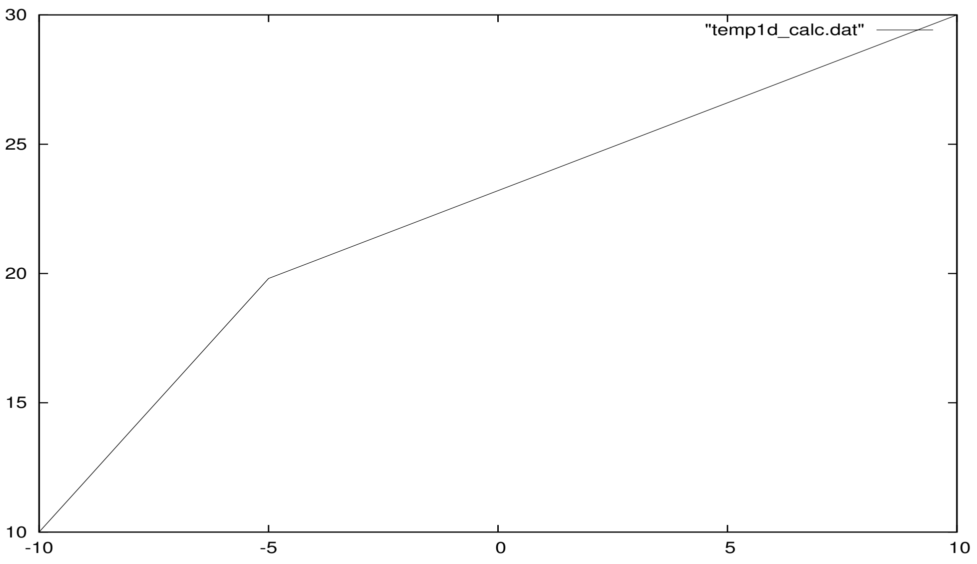
where x' is the location of the unit impulse.

After you have learned how to solve differential equations you will find the theoretical solution to equation (8) to be

$$T(x) = \int_{-L}^{x}\Big[-x\Big(\frac{L+x'}{2kL}\Big) + \frac{L+x'}{2k}\Big]Q(x')dx' + \int_{x}^{L}\Big[x\Big(\frac{L-x'}{2kL}\Big) + \frac{L-x'}{2k}\Big]Q(x')dx' +$$

$$x\Big(\frac{T_2-T_1}{2L}\Big) + \frac{T_2+T_1}{2} \tag{8}$$

For this example we will assign a numerical value of 500 $J/m^3 \cdot sec$ for the value of the source, $Q(x')$, and let k have a value of 390 $W/m \cdot K$. Let the distance, L, be 10 meters. The ends of the bar are held at the constant temperatures $T_1 = 10\,C$, and $T_2 = 30\,C$. We will place the "point" source at the location $x' = -L/2$ on the bar.



The above figure shows the result for this problem when using equation (8) and plotted using GNUPLOT. We will now solve this problem using the HMD cell grid solver and compare the results.

We will now set up a model script for HMD to solve the above temperature distribution problem.

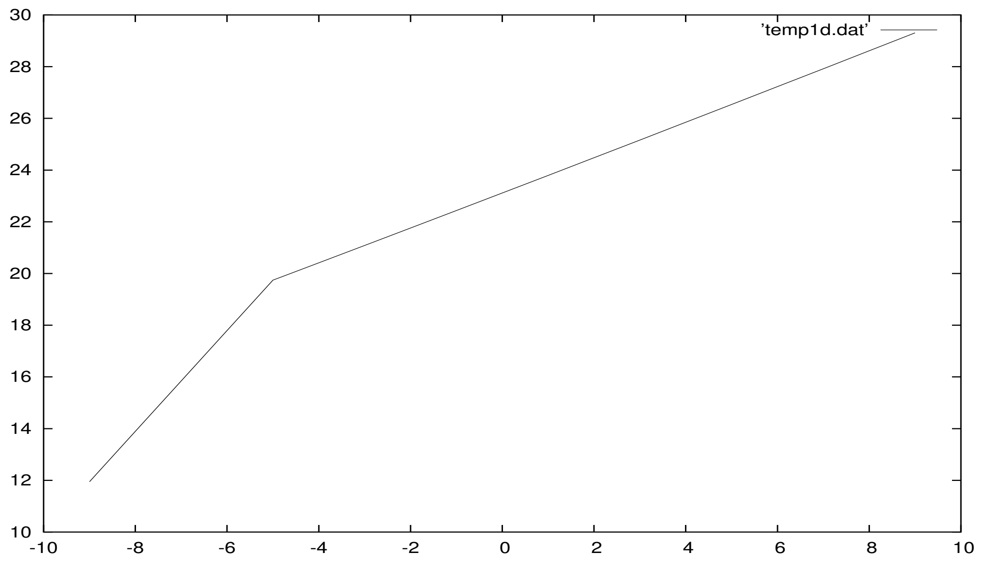
```
//
//   TEMP1D.HMD
//
//
//   User's manual example
//   A 1D bar with a temperature distribution
//   -L <= x <= L, L = 10 meters
//   Temperature held at T1 = 10C at the left end
//   Temperature held at T2 = 30C at the right end
//   A "point" source of 500 J/m^3 * sec at x = -5
//
int ic;
//
//   a force callback
//   impulse at location -L/2, where L = 10
//
function fc1 {double x} {
    if (x > -5.01 && x < -4.99) {
        return -500;
    }
    return 0;
}
//
//   declare a Divergence
//
```

```
//
//   divergence with point force
//   1. Name of cell (to become a chain)
//   2. :Y means 2nd-order differential operator
//   3. / is the "constraint" symbol (the right-hand side)
//
psi :Y ∧ fc1, 1, $1;
//
//   set BC's
//   1. Name of the cell
//   2. _— is the "anchor" symbol used for boundary conditions
//   3. List of BC / Stiffness values for each direction
//        The order is BC_left, stiffness_left, BC_right, stiffness_right
//        and repeated for each direction.
//        LEFT-HAND side
//        a. 10 C is the value at the left boundary
//        b. 390 is the value of all weights (in this case, thermal conductivity)
//        RIGHT-HAND side
//        c. 30 C is the value at the right boundary
//        d. 390 is the value of all weights (in this case, thermal conductivity)
//
psi _| 10, 390, 30, 390;
//
//
//
//   grow layers
//
ic = 0;
while (ic < 9) {
     psi ->;
     ic = ic+1;
}
//
// "link cells"
//
>< psi;
//
//   deduce the cell value
//
?? psi;
//
//   solve
//
double errval;
ic = 0;
errval = 1.0e9;
while (errval > 0.0001 && ic < 1000) {
     >< psi;        // link
     ?? psi;        // deduce
     errval = cellaccuracy(psi);
     ic = ic+1;
}
msgprint 'after link and deduce, ', ic, ' iterations';
```

```
        //
        //   extract the cell data as a plottable data set
        //
        //   args:
        //   1. Name of cell list
        //   2. Name of array that will be created to hold the values
        //   3. Flag that when nonzero creates the x-axis data
        //   4. Dimension number (direction 1=x, 2=y, 3=z, etc)
        //   5. Starting index for line data
        //   6. Ending index for line data
        //   7...N The list of direction indices specifying one point on the line
        //   Since this is 1-dimensional, there is only one index–we choose 0
        //
27      getcell_line psi, 'vpsi', 1, 1, -9, 9, 0;
        //
        //   save the data to a file
        //
        //   args:
        //   1. Name of output file to create
        //   2...N The list of arrays (assumed to be 1 x N vectors) to form the
        //   columns of the data file
        //
28      save 'vpsi.dat', vpsi_s, vpsi;
        //
        //   print a 2-column data set fro the arrays
        //
29      ic = 1;
30      while (ic <= 19) {
31          msgprint vpsi_s[ic], ' ', vpsi[ic];
32          ic = ic + 1;
33      }
34      msgprint 'The End';
```

Note the we make the sign of Q negative in the callback function $fc1$. The HMD green's function is actually not including the minus sign in equation (7), so we must put it into the solution. The function is defined at line 2 and "declared" on the cell creation line, line 8. The boundary values and the thermal conductivities are assigned at line 9. The reader may be annoyed at the reference made in the above script to "divergence" do to the fact that the differential operator employed is actually the Laplacian.

It should be emphasized that the HMD script above provides no display of the data result. The data is saved to a disk file using the *save* command. In this case the output file will consist of two columns: the x coordinates and the potential values. The resulting data are plotted with the GNUPLOT program in the following figure. Comparing this figure with the previous figure obtained from the theoretical result we see that the boundary values are missing in this plot. This is because the boundary values are not themselves node values in the HMD algorithm.


'temp1d.dat'
30
28
26
24
22
20
18
16
14
12
10
-10
-8
-6
-4
-2
0
2
4
6
8
10

Chapter 5

The Finite Element Solver

If you already know how the finite element method works, then all you need to know is how HMD scripts are set up. HMD reads an input script file containing the commands necessary for building and solving the matrix equation for the model. HMD is run from the console command line with the following syntax:

```
hmd    -f    hmd_script_file
```

When you invoke HMD it is assumed that you have already built the mesh, and the mesh file name is given as a parameter in the HMD script file. Alas, building the mesh is half of the job, and this is done with an entirely separate program called GMSH. GMSH is written and distributed by Christophe Geuzaine and is freely available for download at www.geuz.org. The output from GMSH will be a file with a .msh extension, and this file is an input to HMD. In the examples included with HMD you will find files grouped in threes: a .geo file, a .msh file, and a .hmd file. The .geo file is the input file for GMSH, and the result of running GMSH is the .msh file. The .msh file and the .hmd file are both inputs for HMD. There is one important caveat when building the GMSH .geo file: the physical region numbers must be renumbered from 1 to N (see the examples).

Consider the following simple physical problem as an illustration of HMD modeling. Take a stretched string with a static force applied somewhere between its ends.

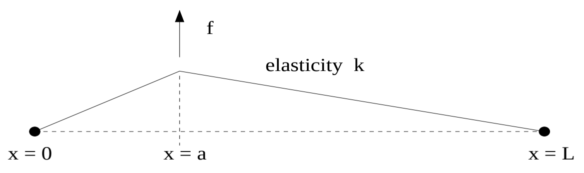


Fig. 1 Stretched string with applied force

We state without proof the mathematics of this problem, which is a second-order, inhomogeneous partial differential equation in one dimension:

$$k\frac{d^2u}{dx^2} = f \tag{1}$$

$$u(0) = 0 \tag{2}$$

$$u(L) = 0 \tag{3}$$

In the finite element method equation (1) is descretized into a set of difference equations assembled into one big matrix equation:

$$\widehat{S} \cdot \vec{u} = \vec{f} \tag{4}$$

In equation (4), $\widehat{S}$ is the numerical equivalent of the second-order derivative called a *stiffness* matrix, and $\vec{f}$ is the numerical equivalent of f called the force vector. We will create the stiffness matrix with the command **buildglobalgrad** and create the vector with **buildglobalvector**. Then we will call **SOLVE** to obtain the solution vector $\vec{u}$.

The geometry is so simple that we can build the .geo file for GMSH without the graphical front end. Specify two points for the ends, make a line, and define regions for the ends (the boundaries) and the line (the material).

```
Point(1) = { 0.0, 0.0, 0.0, 0.1 };
Point(2) = { 10.0, 0.0, 0.0, 0.1 };
Line(1) = { 1, 2 };
Physical Point (1) = { 1 };
Physical Point (2) = { 2 };
Physical Line(3) = { 1 };
```

The above listing for a .geo file shows quite simply the structure of a GMSH input file. However, it will be noticed that there is yet no way to specify the region where the force will be applied. So we will modify the above listing by creating three line segments instead of just one, and the one "Physical Line" statement will become two statements: one for identifying the force region and one for the unforced region.

```
Point(1) = { 0.0, 0.0, 0.0, 0.1 };
Point(2) = { 3.0, 0.0, 0.0, 0.1 };
Point(3) = { 3.2, 0.0, 0.0, 0.1 };
Point(4) = { 10.0, 0.0, 0.0, 0.1 };
Line(1) = { 1, 2 };
Line(2) = { 2, 3 };
Line(3) = { 3, 4 };
Physical Point (1) = { 1 };
Physical Point (2) = { 4 };
Physical Line(3) = { 1, 3 };
Physical Line(4) = { 2 };
```

Now there are four regions: one for each endpoint, one for the string, and one for the small segment of string where the force is applied. Now, to create the mesh file we can type at the console command line

gmsh -1 *geo_file*

where the "-1" argument signifies one-dimensional elements. GMSH will produce a .msh file which will be input to HMD. The mesh file will contain a list of all the nodes along the line created by GMSH, and below that will be the list of all the line elements that GMSH created. Each line element will have specified the nodes that belong to it and the region number in which the element resides.

The HMD script consists of commands that perform the well-known steps for creating the finite element model. We assign physical constants to the elements belonging to the separate regions; we assemble the elements into a global matrix and a global vector; we insert boundary values into the matrix and vector; we solve the matrix equation. That's it. HMD is essentially very simple. Consider the following commands that are used in the building of this example model:

setregionbc 'string', 1, 'polynomial', 0.0 ;

setregiongrad 'string', 3, 7.41 ;

setregionforce 'string', 4, 2.73 ;

buildglobalgrad 'string', 'my_stiffness_matrix' ;

buildglobalvector 'string', 'force_vector' ;

insertbcforce string ;

insertbcmatrix string ;

SOLVE string ;

vec2file 'string', 'string_data.dat' ;

The HMD commands are more or less just procedures, each one grouping together the steps that you would have to program by hand in FORTRAN or Matlab. There is a similar format to the commands, being a descriptive command name, the name of the model, an identifier, and a value.

In addition, there are a few details that are needed which merely support the command syntax. HMD is like a programming language; it has variables, statements, commands, and functions. Unlike FORTRAN and Matlab, variables must be declared by their type before being used–this is like C. The most important variables are the model itself and the mesh. These variables are data structures that hold all the relevant information needed to solve the model. Consider the following variable declarations:

int my_integer ;

mesh string_mesh ;

model string ;

A declaration consists of a reserved keyword followed by the name it will have followed by a semicolon. In the above listing, **model** is a keyword which defines *string* as a model variable. Then the word *string* will be used in all subsequent commands that initialize this model. Always remember to define the mesh variable before you define the model variable. The mesh is considered a more fundamental, more general structure than the model. Several models may be attached to the same mesh. To this end, we must declare the model not only with a name but with the name of the mesh.

model string, 'string_mesh' ;

Finally, there is some top-level information that need to be given at the top of the script. HMD must be told that this model will be solved by the finite element method.

string.modeltype =   FINITE_ELEMENT ;

And the solver must know what kind of matrix equation is to be solved.

string.equation =   POISSON ;

The software distribution comes with a directory called "examples" which includes example scripts. After you look through a few of them you will see that they all follow the same steps.

To summarize, these are HMD modeling steps for following the finite element recipe.

1. Do some initial setup

Declare the mesh, give it a name, a type, and a file name.

Declare the model, give it a name, a type, and an equation.

2. Read the mesh file

loadmesh

3. Assign Boundary conditions

setregionbc

setregionbcn

4. Assign Material coeficients

setregiongrad sets the stiffness values

setregionmass sets the density values

setregionforce sets the force, charge, source term

5. Build the GlobaL Matrix and the Global Vector

buildglobalgrad builds the global stiffness matrix (associated with the gradient term)

buildglobalmass builds the global mass matrix (associated with the time derivative)

buildglobalvector builds the global forcing vector

6. Insert the Boundary Conditions into the Matrix and Vector

insertbcforce puts BCs into the global vector (must always be done first)

insertbcmatrix

7. Solve the Matrix Equation

SOLVE

8. Save the Answer to a File

vec2file saves the vector of nodal values to a file *gmvwrite* saves the vector of nodal values to a GMV file

Chapter 6

Finite Element Examples

1. Poisson's Equation

1-1. 1-D Electrostatics – Dielectric Layers

1-2. 2-D Electrostatics – Concentric Cylinders

2. Helmholtz Equation (wave equation)

2-1. 1-D Normal Modes – Stretched String

2-2. 3-D Normal Modes – Fluid Cylinder

§1. 1-D Electrostatics – Dielectric Layers.

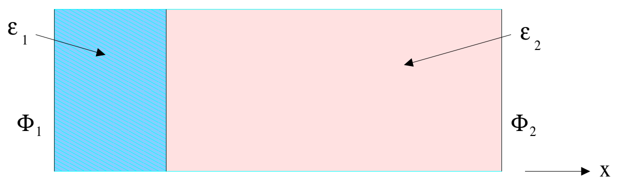


Figure 1. Two dielectric layers

As an example, let us solve a simple one-dimensional problem that can be easily solved analytically as well. Consider two dielectric layers that are bounded in the horizontal, the $\vec{x}$, direction by thin, conducting plates. The media extend without bound in the vertical direction. Because there will be no variation in the vertical direction, the laplacian operator reduces to a second-order derivative in x

$$\nabla^2 \longrightarrow \frac{d^2\phi}{dx^2}$$

We construct the problem so that the end plates are held at a constant potential difference by an external battery and there are no fixed charges in the dielectrics. The potential at $x = 0$ will be V_1, and the potential at $x = L$ will be V_2. Let the position of the boundary between the two different dielectric media be x_d. Our problem is to find the potential, V_d, at this boundary.

Within each region, individually, the gradient of the potential is a contant. To see this, evaluate $\nabla \cdot \vec{D}$ separately in each region (that is, not including the dielectric interface). In region 1,

$$\nabla \cdot \vec{D} = 0 = \nabla \cdot (\epsilon_o \vec{E}_1 + \vec{P}_1)$$

and since the dielectrics are uniform, then $\nabla \cdot \vec{P}_1 = 0$. Therefore,

$$\nabla \cdot \vec{E}_1 = 0$$

$$\frac{d\phi_1}{dx} = const = \frac{V_d - V_1}{x_d} \tag{1}$$

similarly, for region 2,

$$\nabla \cdot \vec{E}_2 = 0$$

$$\frac{d\phi_2}{dx} = const = \frac{V_2 - V_d}{L - x_d} \tag{2}$$

The potential as a function of x in region one can be found by integrating equation (1).

$$d\phi_1 = \frac{V_d - V_1}{x_d} dx$$

$$\int_{V_1}^{\phi_1} d\phi_1 = \frac{V_d - V_1}{x_d} \int_0^x dx$$

$$\phi_1(x) = \frac{V_d - V_1}{x_d} x + V_1 \tag{3}$$

The potential for region 2 can be found from integrating equation 2,

$$\phi_2(x) = \frac{V_2 - V_d}{L - x_d}(x - x_d) + V_d \tag{4}$$

The formulas for the two potentials in equations 3 and 4 depend on the value of the potential V_d. To solve for V_d we will evaluate the equation $\nabla \cdot vecD = 0$ at the interface, x_d. $\vec{D}$ is a macroscopic field whose field lines are defined as beginning and ending on free (not polarization) charge. Unlike $\vec{E}$, which is discontinous at the dielectric interface, $\vec{D}$ is defined to be continous across the boundary. A Gauss law pillbox enclosing the boundary interface in which the x dimension approaches zero yields the equality of $\vec{D}$ on each side of the boundary. At the boundary interface, x_d, we have

$$(\vec{D}_1 - \vec{D}_2) \cdot \hat{n} = 0$$

$$D_1 = D_2$$

$$\epsilon_1 E_1 = \epsilon_2 E_2$$

$$\epsilon_1 \frac{d\phi_1}{dx} = \epsilon_2 \frac{d\phi_2}{dx}$$

$$\epsilon_1 \frac{V_d - V_1}{x_d} = \epsilon_2 \frac{V_2 - V_d}{L - x_d} \tag{5}$$

After doing the algebra and defining $p_a = \epsilon_1/x_d$ and $p_b = \epsilon_2/(L - x_d)$ we can write the simple formula

$$V_d = \frac{p_a V_1 + p_b V_2}{p_a + p_b} \tag{6}$$

We will choose the following physical values and enter them into these derived formulas, equations (3), (4), and (6).

L = 0.6 meters
$x_d = 0.15$ m
$\epsilon_1 = 5.1$
$\epsilon_2 = 2.2$
$V_1 = 1$ volt
$V_2 = 10$ volt

Inserting these values into equation (6) gives for V_d,

$$V_d = 2.1314$$

The formulas (3) and (4) can be used to plot the values of ϕ throughout the dielectric. This is shown in figure 2.

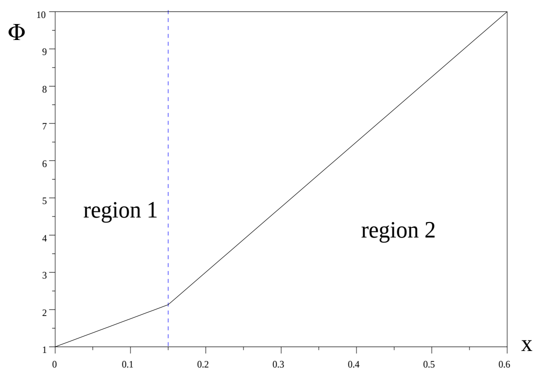


Figure 2. Calculated potential

Now let us build an HMD model for the same problem and compare the results. This will be quit simple to set up because this is essentially a one-dimensional model: there is no variation in the vertical direction. To construct the mesh we merely need a straight line divided into two sections. The mesh file is created using the GMSH program, written by Christophe Geuzaine, and freely available for download at www.geuz.org. GMSH needs a geometry file as input, which we can build by using GMSH's graphical inteface or by hand with our text editor. The following small file took a few minutes with a text editor to create.

```
//
// FE22.GEO
//
// 1d model – 2 dielectric layers
//
scale = 0.02;
x0 = 0.0;
xd = 0.15;
L = 0.6;


//
// define the boundary and interface points
//
Point(1) = x0,0,0,scale; // left end-point
Point(2) = xd,0,0,scale; // interface
Point(3) = L,0,0,scale; // right end-point


//
// define lines that join them
//
Line(1) = 1,2; // dielectric region 1
Line(2) = 2,3; // dielectric region 2
```

```
//
// ”material” regions
//
// Region 1 – the left boundary
//
Physical Point(1) = 1;
//
// Region 2 – the right boundary
//
Physical Point(2) = 3;
//
// Region 3 – the dielectric 1
//
Physical Line(3) = 1;
//
// Region 4 – the dielectric 2
//
Physical Line(4) = 2;
```

Notice that we defined the point locations for the boundaries and interface , then created lines joining them, then defined region numbers associated with each section. No material values are entered yet. That will be done in the HMD script. Now to create the mesh file we run GMSH with our .geo file as input.

```
gmsh    -1   fe22.geo
```

Now we are ready to write the HMD script file. In the HMD script we specify the name of the mesh file, fe22.msh, that should be read. We also specify boundary values (degree-of-freedom constraints) and material values, such as the dielectric constants. The following listing shows the HMD script needed to calculate the solution.

```
//
// FE22.HMD
//
//
// A 1-D model.
//
// 2 dielectric layers in the x direction, uniform in y
//
//
//
// declare the mesh
//
mesh tmesh;
tmesh.meshtype = GMSH;
tmesh.filename = 'fe22.msh';
```

```
//
// Declare the model
//
model fe22, ’tmesh’;
fe22.modeltype = FINITE_ELEMENT_E;
fe22.equation = LAPLACE;


//
// read-in the mesh file
//
loadmesh tmesh;


//
// storage flags:
// bit 0: (0x01) sparsity table to file
// bit 1: (0x02) matrices to file(s)
// bit 2: (0x04) vectors to file(s)
// bit 3: (0x08) calc sparsity with cluster
// bit 4: (0x10) calc matrices with cluster
// bit 5: (0x20) double precision
//
createsparsity ’fe22’, 0x00;


//
// BC values — the potental of the end plates
//
setregionbc ’fe22’, 1, 1.0;
setregionbc ’fe22’, 2, 10.0;


//
// The ”stiffness” coefficients (electric permitivity)
//
setregiongrad ’fe22’, 3, 5.1; // dielectric 1, epsilon_sub_1
setregiongrad ’fe22’, 4, 2.2; // dielectric 2, epsilon_sub_2



//
buildglobalgrad fe22;
buildglobalvector fe22;



//
// set the type flag for the solver
//
setdivergence ’fe22’, ’fe22-grad’;
setsource ’fe22’, ’fe22-force’;
```

```
            //
            // put in the BCs
            //
17          insertbcmodel fe22;



            //
            // solve the equation A*X = B
            // and output the resulting solution vector to 'fe22.dat'
            //
            // args:
            // model name
            // output file name
            // solver flags
            // node-selection specifiers
            //
18          solve_div 'fe22', 'fe22.dat', 0x00, 'all';


19          EXIT;
```

Using the above HMD script, the HMD program is run by invoking the following command,

hmd -f fe22.hmd

The result will be a file called fe22.dat. This file consists of five columns and as many rows (lines) as there are nodes in the model. The numbers in the first column are the node numbers. The numbers in the second column are the potential values calculated at each node. The third, fourth, and fifth columns are the x,y, and z coordinates at each node. Examination of the output data shows that HMD gives a result for this single-precision model indistinguishable from the analytical result up to at least four significant digits. Compare the data plotted in figure 3 to the analytical plot in figure 2.

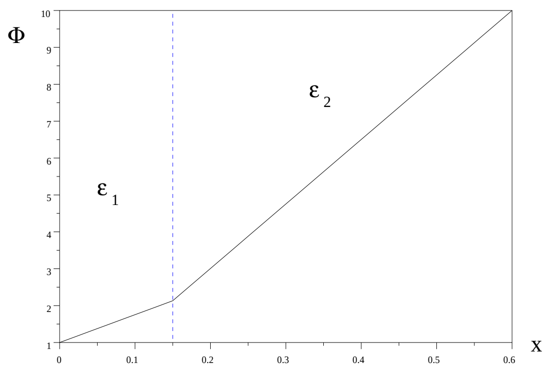


Figure 3. Output data from HMD for dielectric layers

§2. 2-D Electrostatics – Concentric Cylinders.

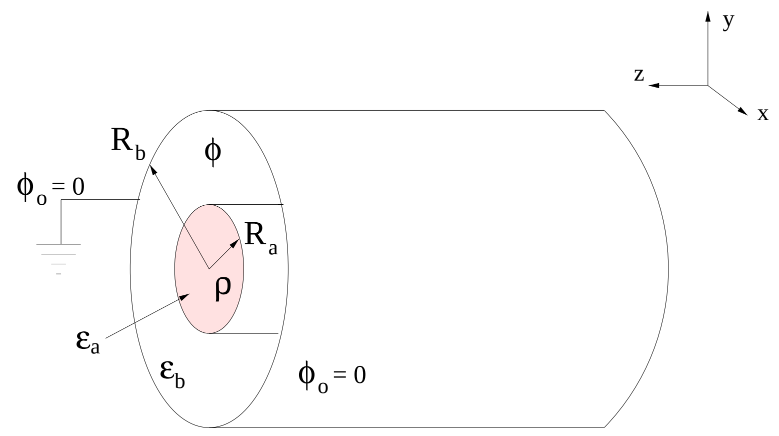


Figure 1. Concentric dielectric cylinders

Consider an infinitely long cylinder containing a cylindrical charge distribution. The outer shell has radius R_b and is held at a electric potential ϕ_o. This could be accomplished if the shell were made of a conducting material, but we will ignore the conductivity for this problem. Let the charge distribution extend to a radius R_a and be denored by ρ. The region included within R_a has a dielectric permitivity of ϵ_a. The region $R_a < r < R_b$ will have a permitivity of ϵ_b. We wish to find the electric potential, ϕ, everywhere within the cylinder $r < R_b$.

Because there is no variation along the axial direction of the cylinder, say the z direction, the derivitives in z are zero. The divergence and laplacian operators reduce to 2-D operators, and so we may take a cross section of the cylinder and treat this as a 2-D problem.

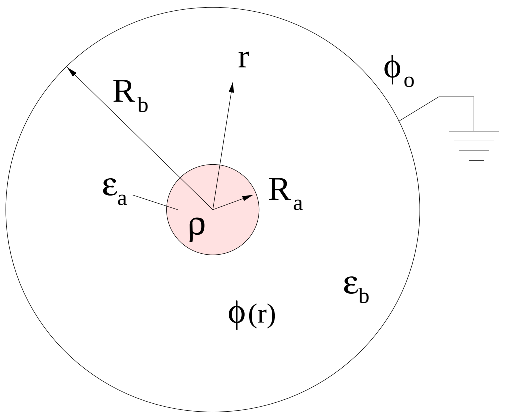


Figure 2. 2-D cross section of concentric cylinders

In order to check the result obtained from the HMD program we will solve this problem analytically. The geometric simplicity of the problem allows the use of the Gauss Law of electrostatics:

$$\oint_S \vec{D} \cdot d\vec{S} = \int_V \rho dV \tag{1}$$

By first obtaining a function for the electric field, $\vec{E}$, we may obtain the electric potential, ϕ, from the property that $\vec{E} = -\nabla\phi$ and integrate $\vec{E}$ through r.

In the two different dielectric regions, the electric field becomes two different functions:

$$\vec{E}_a = \frac{\vec{D}}{\epsilon_a} \tag{2}$$

$$\vec{E}_b = \frac{\vec{D}}{\epsilon_b} \tag{3}$$

We evaluate the Gauss Law formula in region B by integrating $\vec{D}$ over a virtual surface at the arbitrary distance $r_a < r < r_b$. The volume integral on the right side of (1) is integrated only out to r_a. The integral over the length of the cylinder is represented by L, and the integration in the angular direction, θ, is trivially $2\pi r$. we see as well that due to the cylindrical symmetry of the problem that the electric field can only have an r component (the potential does not vary in the θ or z directions). Therefore, we drop the subscript on E_r and call it just E.

$$\epsilon_b E_b L(2\pi r) = \rho L(\pi {r_a}^2)$$

$$E_b(r) = \frac{\rho {r_a}^2}{2\epsilon_b r} \tag{4}$$

Performing a similar evaluation of the Gauss Law within region A, by placing the arbitrary surface of integration at $r < r_a$ results in the following expression:

$$\epsilon_a E_a L(2\pi r) = \rho L(\pi r^2)$$

$$E_a(r) = \frac{\rho r}{2\epsilon_a} \tag{5}$$

The electrostatic potential can be found by integrating the electric field from $r = r_b$, where the potential is grounded, to a point r within the cylinder.

$$\vec{E} = -\nabla\phi$$

$$\phi_2 - \phi_1 = -\int_1^2 \vec{E} \cdot d\vec{l}$$

This is equivalent to the work that must be done to move a point charge toward the region of central charge density from the outer shell. For region B, the integration will be from $r = r_b$ to r.

$$\phi_b(r) = -\int_{r_b}^{r} E_b(r) dr$$

$$\phi_b(r) = \frac{\rho {r_a}^2}{2\epsilon_b} \ln(\frac{r_b}{r}) \tag{6}$$

For region A, we add the potential obtained by integrating from r_b to r_a to the functional integral from r_a to r.

$$\phi_a(r) = -\int_{r_a}^{r} E_a(r) dr + \phi_{ba}$$

$$\phi_a(r) = -\int_{r_a}^{r} \frac{\rho}{2\epsilon_a} r dr + \frac{\rho {r_a}^2}{2\epsilon_b} \ln(\frac{r_b}{r_a})$$

$$\phi_a(r) = \frac{\rho}{4\epsilon_a}({r_a}^2 - r^2) + \frac{\rho {r_a}^2}{2\epsilon_b} \ln(\frac{r_b}{r_a}) \tag{7}$$

Choosing physical values that will scale well numerically, we choose $\epsilon_a = 5.1$ (glass), $\epsilon_b = 2.2$ (polypropylene), $\rho = 10$ (coulombs/m^3), $r_a = 0.1m$, $r_b = 0.5m$. The Scilab plot in figure 3 shows equations (6) and (7) plotted again radius.

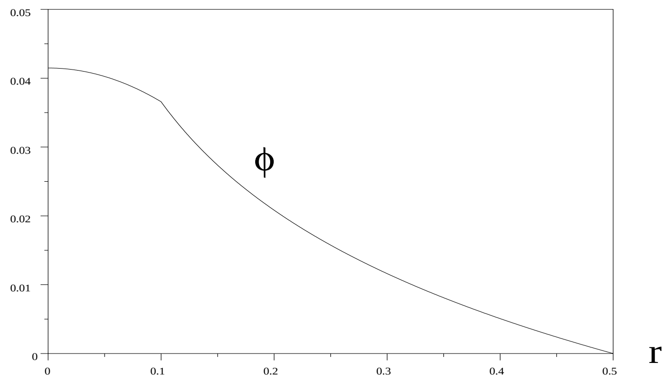


Figure 3. Plot of calculated electric potential

We will now model this 2-D problem using HMD and GMSH. The grid of x,y points and elements is created using the GMSH (www.geuz.org) program. The HMD modeling program is designed to read a GMSH output file (mesh file) and use that to construct its finite elements. The geometry of the problem is entered into a GMSH input file, called a .geo file. From this GMSH produces a mesh file that will be used by HMD.

For GMSH we need to define two concentric circles and (1) tell it that the outer circle is a "region" (the boundary), (2) tell it that the inner disk is a "region" (the charge distribution), and (3) tell it the the outer area, $r_a < r < r_b$, is a "region" (the dielectric). The code given in listing 1 is the GMSH script used to create the mesh.

```
//
// listing 1.      fe19.geo
//
```

```
// 2-D model for Mach II
//

bigradius= 0.5;
smallradius = 0.1;
bigscale = 0.1;
smallscale = 0.02;

//
// Points defining the outer circle
//
Point(1) = {0, 0, 0, smallscale};
Point(2) = {bigradius, 0, 0, bigscale};
Point(3) = {0, bigradius, 0, bigscale};
Point(4) = {-bigradius, 0, 0, bigscale};
Point(5) = {0, -bigradius, 0, bigscale};

//
// Points defining the inner circle
//
Point(6) = {smallradius, 0, 0, smallscale};
Point(7) = {0, smallradius, 0, smallscale};
Point(8) = {-smallradius, 0, 0, smallscale};
Point(9) = {0, -smallradius, 0, smallscale};

//
// Outer circle
//
Circle(1) = {2,1,3};
Circle(2) = {3,1,4};
Circle(3) = {4,1,5};
Circle(4) = {5,1,2};

//
// Inner circle
//
Circle(5) = {6,1,7};
Circle(6) = {7,1,8};
Circle(7) = {8,1,9};
Circle(8) = {9,1,6};

//
// surface of inner disk
//
Line Loop(9) = {5,6,7,8};
Plane Surface(10) = {9};

//
// surface of Outer disk
//
Line Loop(11) = {1,2,3,4};
Plane Surface(12) = {11,9};

//
// Region 1 – the outer circle boundary
```

```
//
Physical Line(1) = {4,1,2,3};

//
// Region 2 – the inner disk
//
Physical Surface(2) = {10};

//
// Region 3 – the outer disk
//
Physical Surface(3) = {12};
```

The GMSH script is listing 1, when compiled with the GMSH program, produces a mesh file called fe19.msh. This is performed with the following command:

```
gmsh    -2   fe19.geo
```

Because this model has circular symmetry, I would suspect that the mesh file generated will have some node numbers missing from the natural sequence 1..N. I think this is because a point was defined for the circle centers which is not included in the mesh. The HMD program expects that the node numbers found in the GMSH mesh file will be in the range 1..N, with no indices missing from the sequence. To be sure that the mesh file conforms to this rule, we will filter the mesh file through an AWK script to reorder node numbers if needed:

```
cat fe19.msh | awk  -f  reorder.awk  fe19.msh > tmp.msh
```

You may then rename the temporary file, tmp.msh, to fe19.msh.

Now we are ready to write the HMD script that will compute the finite element solution. The code listing given in Listing 2 will generate the finite element solution that we desire.

```
//
// Listing 2,   FE19.HMD
//
//
// A 2-D model.
// Two concentric circles. The inner circle has a chage density.
//
//
//
//
// declare the mesh
//
mesh tmesh;
tmesh.meshtype = GMSH;
tmesh.filename = 'fe19.msh';


//
// Declare the model
```

```
//
model fe19, ’tmesh’;
fe19.modeltype = FINITE_ELEMENT_E;
fe19.equation = POISSON;


//
// read-in the mesh file
//
loadmesh tmesh;


//
// storage flags:
// bit 0: (0x01) sparsity table to file
// bit 1: (0x02) matrices to file(s)
// bit 2: (0x04) vectors to file(s)
// bit 3: (0x08) calc sparsity with cluster
// bit 4: (0x10) calc matrices with cluster
// bit 5: (0x20) double precision
//
createsparsity ’fe19’, 0x00;


//
// BC
//
setregionbc ’fe19’, 1, 0;


//
// The ”stiffness” coefficients (electric permitivity)
//
setregiongrad ’fe19’, 2, 5.1; // inner disk, epsilon_sub_a
setregiongrad ’fe19’, 3, 2.2; // outer disk, epsilon_sub_b


setregionforce ’fe19’, 2, 10.0; // inner disk, charge distribution


//
buildglobalgrad fe19;
buildglobalvector fe19;


//
// set the type flag for the solver
//
setdivergence ’fe19’, ’fe19-grad’;
setsource ’fe19’, ’fe19-force’;


//
// put in the BCs
//
insertbcmodel fe19;
```

```
         //
         // solve the equation A*X = B
         // and output the resulting solution vector to 'sol.dat'
         //
         // args:
         // model name
         // output file name
         // solver flags
         // node-selection specifiers
         //
18       solve_div 'fe19', 'sol.dat', 0x00, 'all';


19       EXIT;
```

The HMD script above can be explained in terms of the basic steps required for assembling a finite element model. Lines 1 through 6 are variable declarations, namely of the Mesh structure and the Model structure. The mesh declarations cause HMD to allocate and initialize a structure to hold the mesh coordinates and node identifier numbers. The model structure is a top-level object that allows access to all the parameters that apply to this particular model.

The GMSH program will have created a file called fe19.msh which contains all the coordinates and node numbers (and material region numbers). This information is loaded into HMD on line 7 of the script. Line 8 creates a special structure for holding the sparsity information of the model. Consequently, the matrices produced for the model will be stored internally in a packed format (packed matrices are specified by model type FINITE_ELEMENT_E, while unpacked matrices are specified by FINITE_ELEMENT).

On lines 9 through 12 we enter the material coefficients. To the engineer, these are the material properties and degree-of-freedom (DOF) constrains. To the mathematician, these are the linear coefficients on the differential equation and the boundary values.

Lines 13 and 14 comprise the primary usefulness of HMD. These commands take the mesh information and the material coefficients and construct a matrix for each "element" in the mesh and assemble all the element matrices into one global matrix and one global vector. The idea is that we will solve the matrix equation

$$\widehat{S} \cdot \vec{u} = \vec{f} \tag{8}$$

The command buildglobalgrad constructs the matrix $\widehat{S}$, and buildglobalvector creates the vector $\vec{f}$. Lines 15 and 16 are important only within the contect of the HMD programming langauge. During the model construction the matrices and vectors and given "types", which are essentially states in the process. Here we set the stiffness matrix to have a type DIVERGENCE and the forcing vector to have a type SOURCE. This is like changing to solution mode. (lines 15 and 16 are required for a model of type FINITE_ELEMENT_E, but not FINITE_ELEMENT). Line 17 modifies the global matrix and global vector to contain the contraints or boundary values (which were specified on line 9).

At this point, you may export the constructed matrix and vector to ascii files and use some other software to solve the matrix equation, say Matlab or Scilab or Algae. You would do that with

```
expandmatrix 'fe19', 'fe19-grad', 'gradfile.dat';
```

and

```
expandvector 'fe19', 'fe19-force', 'forcefile.dat';
```

However, HMD has the capability to solve the matrix equation, and this is what is specified on line 18. This command will result in a file called sol.dat which contains the solution. The solution is the value calculated at each node position. The structure of the output file is 5 ascii columns with as many lines as there are nodes in the model. Column 1 is the node number; column 2 is the nodel value (the answer); column 3, 4, and 5 are the x,y, and z coordinates of the node on that line.

The above HMD script, fe19.hmd, is executed with the HMD program using the following command:

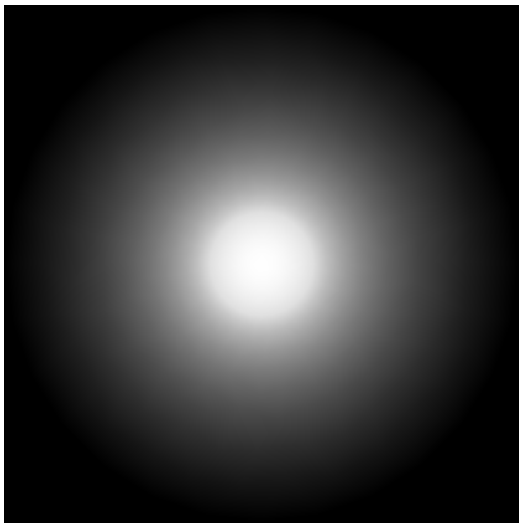
```
hmd -f fe19.hmd
```

The output file is sol.dat, as specified in the HMD script. The output data is in the form of the electric scalar potential evaluated at each of the node points, which is difficult to examine visually. The HMD program comes with an interpolation program, INTP, that creates a matrix of regularly-spaced values. For example, an image of the data may be created using INTP by making a matrix of raw, binary pixels, then converting that raw image into a displayable image (using ImageMagick or your favorite image program):

```
intp -d sol.dat -g fe19.msh -b scalar -o fe19.raw
```

Now, using the ImageMagick "convert" command,

```
convert -size 417x417 -depth 8 gray:fe19.raw fe19.pdf
```



The resulting image above looks more like a long exposure of a star from an astronomical camera, but the larger potential values are shown as the brighter pixels, and the smaller values are the darker pixels. With your favorite image program you may create a false-color image that better identifies the potential distribution. If you would rather use Matlab or Scilab to display the interpolated values, there is a corresponding INTP command,

intp -d sol.dat -g fe19.msh -a scalar -o fe19.mat

Now the file fe19.mat can be read into Matlab with the dlmread command:

M = **dlmread** ('fe19.mat', ' ');

And the cooresponding command for Scilab is

M = **read** ('fe19.mat', -1, 417);

Now that we have the Matlab-able file of data values, we can extract a profile line from the matrix and compare this result to the theoretical calculation displayed in figure 3. From the matrix, M, that was read into Matlab, a profile can be taken at row 208 and columns 208 through 417.

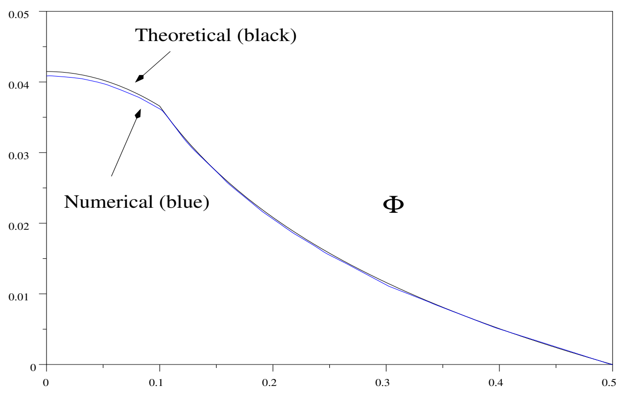


Figure 5. Comparison of results

§3. 1-D Normal Modes – Stretched String.

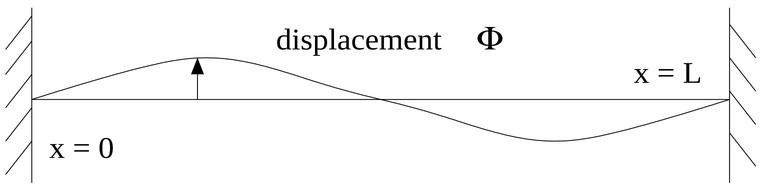


Figure 1. Stretched String

On a string stretched between two points, $x = 0$ and $x = L$, a wave travels to the right, $f_+(x - vt)$. The endpoints are fixed so that the wave is reflected as $f_-(x + vt)$. The combination of the forward and reflected waves results in a standing wave, $F(x, t)$. This standing wave of arbitrary shape may be represented as the superposition of normal modes. A normal mode by definition has the property that every material element moves (vibrates) with the same phase, so the position and time coordinates are independent. This motivates the separation of variables x and t so that each normal mode function is a product of two functions

$$F_\omega(x, t) = \phi_\omega(x) T_\omega(t)$$

.

The form of $\phi_\omega(x)$ is found by the solution by separation of variables of the wave equation in the region $x = 0$ to $x = L$. Without reproducing the derivation, we may state that when the wave equation

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{1}{c^2} \frac{\partial^2 \phi}{\partial t^2}$$

is separated into equations for x and t, the equation in x is the homogeneous Helmholtz equation

$$\frac{\partial^2 \phi}{\partial x^2} + k^2 \phi = 0 \tag{1}$$

where $k = \omega/c$.

The solution to differential equation (1) is $sin(kx)$. The wave number , k, is found by fitting the solution to the boundary coinditions. At $x = L$ the mode function $sin(kx)$ must be zero so that

$$kL = n\pi$$

or

$$k = \frac{n\pi}{L} \tag{2}$$

where n takes on all the integer value from 1 to ∞.

Notice that equation (1) is an eigenvalue equation whose eigenvalues are k^2. Since HMD has an eigenmode solver, we should be able generate the vibrational modes of this string with HMD and compare the result with the known answer. Let the length, L, of the string be 1 meter, and let the sound speed, c, in the material be 200 meters per second. Then the modal frequencies given by equation (2) are given in the following table (in hertz):

n	f
1	100
2	200
3	300
4	400

This is a rather simple model to construct in HMD. The mesh is trivial in that you merely have to draw a line with GMSH and specify the endpoints as separate regions. The following code listing is the GMSH input file used to produce the simple stretched string mesh.

```
//
// FE7.GEO
//
// A stretched string (1-d FE problem)
//


//
// end points
//
Point(1) = 0, 0, 0, 0.01;
Point(2) = 1, 0, 0, 0.01;


//
// the line
//
Line (1) = 1, 2;



//
// Region 1 – the line
//
Physical Line (1) = 1;


//
// LHS point
//
Physical Point (2) = 1;


//
// RHS point
//
Physical Point (3) = 2;
```

The above input file must now be compiled by the GMSH program to produce the mesh file. Since the input file is called fe7.geo, the mesh file will be automatically named by GMSH to be fe7.msh. This mesh file will be an input file to HMD.

```
gmsh    -1    fe7.geo
```

Notice that we use a **-1** as an argument to GMSH. This specifies that the mesh will be one-dimensional. If you were to create the GEO file using the GMSH graphical interface then you would merely press the F1 key to generate a mesh (or the F2 key for a two-dimensional model, or the F3 key for a three-dimensional model).

Now we need an HMD input file. The following code listing is an HMD script that will create the finite element matrix equation for the string normal modes and solve it.

```
//
// FE7.HMD
//
// Solve Eigen Modes equation
// 1-D
// A line in the x-direction, represents a stretched string
//
//
//
//
// declare the mesh first
//
mesh tmesh;
tmesh.meshtype = GMSH;
tmesh.filename = 'fe7.msh';



//
// Declare the model
//
model tmodel, 'tmesh';
tmodel.modeltype = FINITE_ELEMENT;
tmodel.equation = EIGENMODES; // means d2̂U/dx2̂ + k2̂ U = 0


//
// read-in the mesh file
//
loadmesh tmesh;


//
// set physical values for each region, each model
//
```

```
    // ————————————————————————————————————————
    // BC
    // Region 2 – the LHS boundary point
    // Pinning the LHS point, setting the node value to zero.
    //
8   setregionbc ’tmodel’, 2, ’polynomial’, 0.0; // perimeter
    //
    // BC
    // Region 3 – the RHS boundary point
    // Pinning the RHS point, setting the node value to zero.
    //
    //
9   setregionbc ’tmodel’, 3, ’polynomial’, 0.0; // perimeter
    // ————————————————————————————————————————


    //
    // Region 1 – the string itself
    //
    //
    // GRAD
    // The ”grad” coefficient is the stiffness
    // But here, we normalize the equation so that the ”mass” coeficient
    // contains all the physics
    //
10  setregiongrad ’tmodel’, 1, 1.0;
    //
    // MASS
    // The ”mass” coefficient is 1 / c2̂, i.e., reciprocal of sound-speed
    // squared
    //
    // Le the length of the string be 1 meter
    // Let the frequency (fundamental) be 100 Hz
    // The sound speed = 2 * L * f
    // c = 200 m/s
    // 1 / c2̂ = 2.5e-5
    //
11  setregionmass ’tmodel’, 1, 2.5e-5;



    //
    // build the global Gradient matrix
    //
12  buildglobalgrad ’tmodel’, ’grad’;
13  buildglobalmass ’tmodel’, ’mass’;


    //
    // apply boundary conditions
    //
    // this function applied BC’s to the Grad and Mass
    // matrices separately–because the LAPACK routine
    // for solving it will take them separately, not combined.
```

```
        //
14      insertbceigen tmodel;


        //
        // solve the FE matrix
        //
15      SOLVE tmodel;


        //
        // output the eigenvalues
        //
16      eigenvalueswrite 'tmodel', 'fe7-vals.dat';
        //
        // output the eigenvectors (each column is a vector)
        //
17      eigenvectorswrite 'tmodel', 'fe7-vecs.dat';


        //
        // (the node indices corresponding to each row in these files
        // will be automatically saved in "tmodel-packlist.dat"
        //


        // end
```

It should be pointed out that there are a few code statements in this file that seem inconsistent with the other code listings that were presented. Notice that the model type is *FINITE_ELEMENT* not *FINITE_ELEMENT_E* as was used in previous examples. This tells you that the above script is using HMD's original, primitive algorithm that I call "mach I". Although this algorithm is no longer under development, it works, and I don't plan to remove it as it serves as a check when doing diagnostics. Other differences are that there is no *createsparsity* statement; the *setregionbc* statement has a "polynomial" specifier that is not needed in the updated algorithm; the updated algorithm always uses *insertbcmodel* whereas here we have *insertbceigen*; the updated algorithm specifies the output file name in the *solve_eigen* command whereas here we have the *SOLVE* command followed by two separate commands for writing output files for the eigenvectors and eigenvalues. This "mach I" algorithm has only limited capability: it has only single precision; you can't choose which LAPACK solver routine to use with a bit-field flag as in the updated algorithm.

We will now solve the model by issuing the following command at the command prompt:

```
hmd -f fe7.hmd
```

The output file generated are

```
fe7-vals.dat
fe7-vecs.dat
tmodel-packlist.dat
```

The first two files contain the eigenvalues and eigenvectors, respectively. The third file is produced only when using the old, mach I algorithm. It contains a list of node numbers that are the cross reference into the file of eigenvectors. When boundary values are set to zero, the corresponding node positions are removed from the global matrix since these rows and columns would merely produce unit eigenvectors. We keep track of which node positions correspond to components in the eigenvector with the packlist.

The eigenvalues file, fe7-vals.dat, is easy to decipher as it is a single column ascii file. The eigenvalues contained in this file are the angular frequency squared. To convert to frequency in hertz you must take the square root and divide by 2π. The following table shows the first few eigenvalues and the corresponding frequencies obtained from the HMD model.

ω^2	f
394859.0	100.009476
1580040.0	200.057194
3556280.0	300.136010
6325540.0	400.284649

The first four mode shapes that are output from HMD are shown in figures 2 through 5. We recognize the familiar sin function.

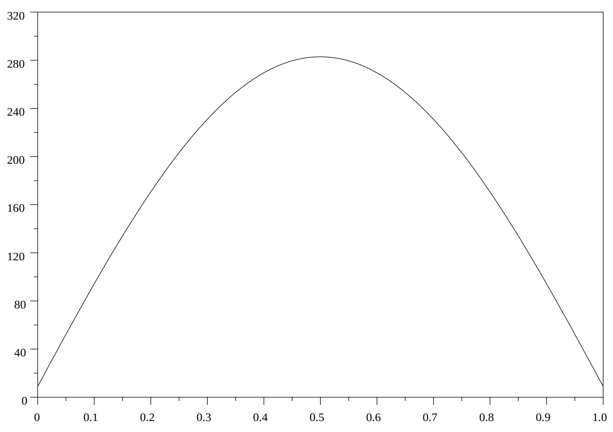


Figure 2. Mode 1 from column 1 of fe7-vecs.dat

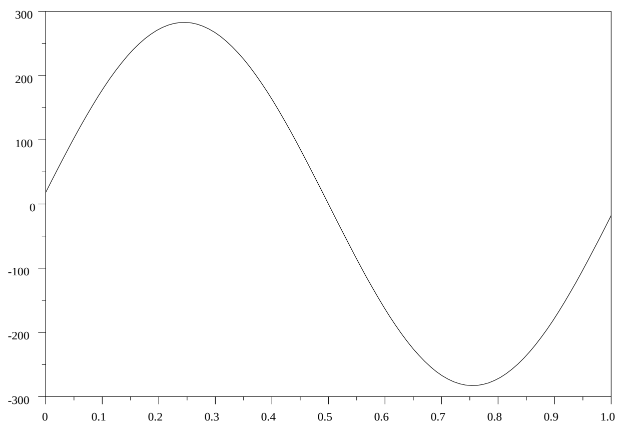


Figure 3. Mode 2 from column 2 of fe7-vecs.dat

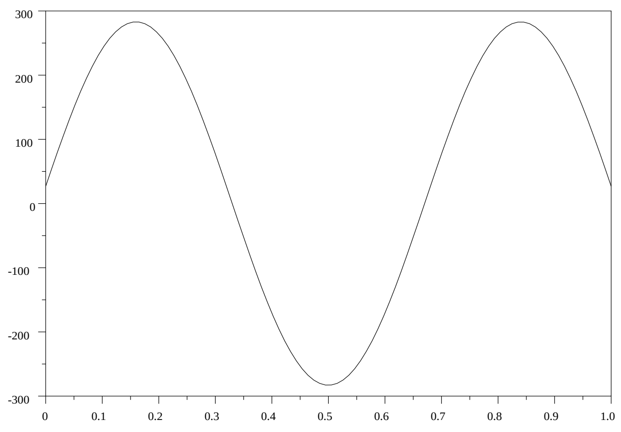


Figure 4. Mode 3 from column 3 of fe7-vecs.dat

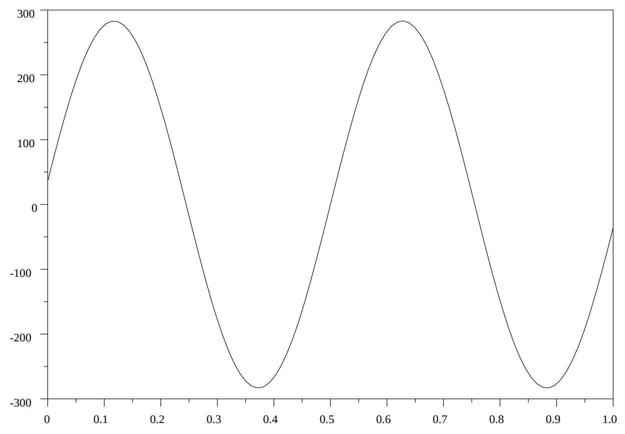


Figure 5. Mode 4 from column 4 of fe7-vecs.dat

§4. 3-D Normal Modes – Fluid Cylinder.

Consider the resonant modes inside a cylinder filled with fluid.

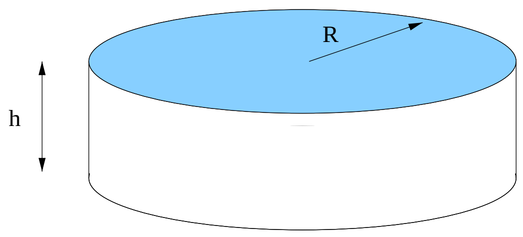


Figure 1. Cylinder filled with fluid.

The scalar variable will be the velocity potential defined by $\vec{v} = -\nabla\phi$. We then solve for the normal modes of this scalar potential. For the calculation of mode frequencies it is immaterial that this scalar potential is an artificial quantity. But for the specification of the boundary conditions we must discover how ϕ is related to the pressure, since pressure the wave equation variable for a fluid.

The conservation of momentum equation for a fluid (just Newton's Second Law) is

$$\rho\frac{\partial\vec{v}}{\partial t} = -\nabla p \tag{1}$$

where ρ is the static fluid density and p is the acoustic pressure. Substituting $\vec{v} = -\nabla\phi$ into equation (1) gives

$$\rho\frac{\partial(-\nabla\phi)}{\partial t} = -\nabla p$$

interchanging the order of differentiation on the left gives

$$-\rho\nabla(\frac{\partial\phi}{\partial t}) = -\nabla p$$

Since ρ is a zeroth-order quantity, it is considered constant with respect to the spacial differentiation. So we may put it inside the gradient operation giving

$$-\nabla(\rho\frac{\partial\phi}{\partial t}) = -\nabla p$$

and now integrating both sides gives

$$p = \rho\frac{\partial\phi}{\partial t} \tag{2}$$

The normal mode problem is actually in frequency space, as we are solving the homogeneous Helmholtz equation

$$\nabla^2\phi + k^2\phi = 0 \tag{3}$$

The ϕ in equation (3) is therefore $\phi(\vec{r},\omega)$, so the time derivative in equation (2) is transformed to frequency space by

$$\frac{\partial\phi}{\partial t} = -j\omega\phi \tag{4}$$

Then substituting equation (4) into equation (2) we obtain the relation between the acoustic pressure and the scalar potential,

$$p = -j\rho\omega\phi \tag{5}$$

Therefore, when we set homogeneous dirichtlet boundary conditions we are really setting the pressure to zero, and when we set homogeneous Neumann boundary conditions we are setting the fluid velocity to zero.

The normal mode frequencies are found analytically by solving equation (3) in cylindrical polar coordinates. The procedure is to substitute the ∇^2 operator with its cylindrical coordinate equivalent operator, then perform separation of variables. The following result was taken from Arfken, *Mathematical Methods for Physicists*, 2nd edition, page 635.

$$\phi = \Sigma_{mnk}\ \ J_m(\frac{\alpha_{mn}}{a}r)e^{\pm jm\theta}(a_{mn}\sin(kz)\ \ or\ \ b_{mn}\cos(kz)) \tag{6}$$

The index m identifies the dimetral mode, the index n identifies the radial mode, and the index k corresponds to modes in the z direction. When $r = a$ we want the radial mode shape, J_m to be zero. So the α_{mn} are the zeros of J_m given by

n	$J_0(x)$	$J_1(x)$	$J_2(x)$	$J_3(x)$
1	2.4048	3.8317	5.1356	6.3802
2	5.5201	7.0156	8.4172	9.7610
3	8.6537	10.1735	11.6198	13.0152

At $z = 0$ and $z = h$ we want ϕ to vanish, so in equation (6) we choose the axial mode shape to be $sin(kz)$. This boundary condition forces the values of k to be

$$k = \frac{p\pi}{h}$$

where $p = 0, 1, \ldots$. For homogeneous, dirichtlet boundary conditions, the mode frequencies are given by (see Arfken)

$$\omega_{mnp} = c\sqrt{\frac{{\alpha_{mn}}^2}{a^2} + \frac{p^2\pi^2}{h^2}} \tag{7}$$

where c is the sound speed.

Let the sound speed be set to $c = 200m/s$, the radiua $a = 12m$, and the height $h = 6m$. We can use equation (7) and table (1) to calculate what the first few frequencies should be. The magnitudes of the Bessel zeros in table (1) will govern the order in which the eigenvalues will be found. The values of the index p in equation (7) will start at 1, not zero. This is because we are pinning the boundary value to zero at

$z = 0$ and $z = L$: therefore there can be no pure translation mode in the z direction. The first frequency (in radians) is $\omega_{011} = 112.128$, the second will be $\omega_{111} = 122.656$, the third will be $\omega_{211} = 135.250$, and the fourth is $\omega_{021} = 139.393$.

Now let us use HMD to solve this problem and compare its answer with the theoretical solution. Building a cylinder in GMSH is a simple matter. The following GEO file is the input to the GMSH program that will produce the necessary MSH file:

```
//
// fe23.geo
//
// Cylinder, 3D
//


scale = 1.7;
radius = 12;
z1 = 0;
z2 = 6;

//
// bottom disk
//
Point(1) = 0, 0, z1, scale; // center
Point(2) = radius, 0, z1, scale;
Point(3) = 0, radius, z1, scale;
Point(4) = -radius, 0, z1, scale;
Point(5) = 0, -radius, z1, scale;

//
// top disk
//
Point(6) = 0, 0, z2, scale; // center
Point(7) = radius, 0, z2, scale;
Point(8) = 0, radius, z2, scale;
Point(9) = -radius, 0, z2, scale;
Point(10) = 0, -radius, z2, scale;

//
// arcs for bottom disk
//
Circle(1) = 2,1,3 Plane0,0,1;
Circle(2) = 3,1,4 Plane0,0,1;
Circle(3) = 4,1,5 Plane0,0,1;
Circle(4) = 5,1,2 Plane0,0,1;

//
// arcs for top disk
//
```

```
Circle(5) = 7,6,8 Plane0,0,1;
Circle(6) = 8,6,9 Plane0,0,1;
Circle(7) = 9,6,10 Plane0,0,1;
Circle(8) = 10,6,7 Plane0,0,1;


Line(9) = 5,10;
Line(10) = 2,7;
Line(11) = 3,8;
Line(12) = 4,9;




//
// surface around the rim
//
Line Loop(13) = 10,-8,-9,4;
Ruled Surface(14) = 13;
Line Loop(15) = 11,-5,-10,1;
Ruled Surface(16) = 15;
Line Loop(17) = 12,-6,-11,2;
Ruled Surface(18) = 17;
Line Loop(19) = 9,-7,-12,3;
Ruled Surface(20) = 19;


//
// top and bottom surface
//
Line Loop(21) = 8,5,6,7;
Plane Surface(22) = 21;
Line Loop(23) = 4,1,2,3;
Plane Surface(24) = 23;
//
// the volume
//
Surface Loop(26) = 22,14,16,18,20,24;
Volume(27) = 26;




//
// Region 1 – the rim
//
Physical Surface(1) = 14,16,18,20;


//
// Region 2 – top and bottom
//
Physical Surface(2) = 22,24;


//
```

```
// region 3 – the volume
//
Physical Volume(3) = 27;
```

To create the MSH file we must run GMSH, specifying the above file as its input. The following command will run GMSH and create the MSH file:

```
gmsh    -3    fe23.geo
```

Now we create the HMD script file. The following script file tells HMD to use the MSH file we just created and generate the finite element solution.

```
//
// FE23.HMD
//
// Solve Eigen Modes equation
// 3-D
// A cylinder
//
//
//
//
// declare the mesh first
//
mesh tmesh;
tmesh.meshtype = GMSH;
tmesh.filename = 'fe23.msh';



//
// Declare the model
//
model tmodel, 'tmesh';
tmodel.modeltype = FINITE_ELEMENT_E;
tmodel.equation = EIGENMODES;


//
// read-in the mesh file
//
loadmesh tmesh;


//
// set physical values for each region, each model
//
createsparsity 'tmodel', 0x00;
```

```
// ————————————————————————
// BC
// Region 1
// Pinning the boundary, setting the node values to zero.
//
setregionbc 'tmodel', 1, 0.0; // perimeter
setregionbc 'tmodel', 2, 0.0; // top and bottom
// ————————————————————————


//
// Region 3 – the cavity itself
//
//
// GRAD
// The "grad" coefficient is the stiffness
// But here, we normalize the equation so that the "mass" coeficient
// contains all the physics
//
setregiongrad 'tmodel', 3, 1.0;
//
// MASS
// The "mass" coefficient is 1 / $c^2$, i.e., reciprocal of sound-speed
// squared
//
setregionmass 'tmodel', 3, 2.5e-5;



//
// build the global Gradient matrix
//
buildglobalgrad 'tmodel', 'grad';
buildglobalmass 'tmodel', 'mass';


setdivergence 'tmodel', 'grad';
seteigenmatrix 'tmodel', 'mass';


//
// apply boundary conditions
//
// this function applied BC's to the Grad and Mass
// matrices separately–because the LAPACK routine
// for solving it will take them separately, not combined.
//
insertbcmodel tmodel;


//
// solve the FE matrix
//
// flags:
        // bits 0-3: 0= use LAPACK, symmetric, generalized eigenvalue solver
        // bit 4: 1= use symmetric-packed solver (use upper triamgle)
        // bit 5: 1= use SPARSPAK band-packing of sparse matrix
        // bit 7: 1= don't calculate eigenvectors
        //
18      solve_eigen 'tmodel', 'fe23.dat', 0x10, 'all';




        //
        // (the node indices corresponding to each row in these files
        // will be automatically saved in "tmodel-packlist.dat"
        //
19      EXIT;
        // end
```

In the above script listing for FE23.HMD, notice the bit flags argument to the solver in line 18. Here we use bit 4, which in hexidecimal is 0x10. This flag specifies that we will use a symmetric-packed solver and so (in principal) reduce the memory requirements. This algorithm works on global matrices that are symmetric, which will usually be the case for a simple finite element problem with no damping. Since the matrices are symmetric, we only need the upper triangle, thereby reducing the memory requirements by half. However, in the above script the memory requirements have not really been reduced because all the eigenvectors have been requested. The solver must allocate a full N x N matrix to hold the eigenvectors. Had we desired to calculate a solution with the eigenvalues only and no eigenvectors, we would have set bit 7 in the flags argument. The hexidecimal representation of the flags would then have been 0x90.

The memory requirements could have been reduced still further had we set bit 5 instead of bit 4. This algorithm actually rearranges the global matrices to put all the nonzero matrix elements near the diagonal so that a very efficient packing can be accomplished. If you try this you will find extra ones in the list of eigenvalues. These are part of the unit vector space that results when homogenous boundary values are given. The other algorithms remove the unit vector space by default, making the output easier to read.

To run the above script, type the following command at the system command prompt:

```
hmd -f fe23.hmd
```

The output data are written to the file fe23.dat. The first column in this file is comprised of the node numbers. Unlike the Poisson solver, the node numbers here are not important in relation to the eigenvalues. This is because the LAPACK eigenvalue equation solver outputs the eigenvalues in increasing order, which is not necessarily the same order as the node numbers. However, the node numbers do in fact correspond to the vector component locations so that the eigenvector components can be assigned to the appropriate spacial locations when doing graphics on them.

The second column in the file contains the eigenvalues. These values are equivalent to ω^2. In order to obtain the frequency in hertz you must take the square root and divide by 2π. Columns 3 through 5 are the x, y, and z coordinates belonging to each node number in column 1. Starting in column 6 the mode vectors (eigenvectors) are listed. Column 6 is mode 1, column 7 is mode 2, etc.

Comparing the output from HMD with the theoretical result calculated above we obtain the following table of values (as ω):

eigenvalue (ω)	Theoretical	HMD output	deviation (%)
ω_{011}	112.128	116.450	3.9
ω_{111}	122.656	128.69	4.9
ω_{211}	135.250	143.118	6.3
ω_{021}	139.393	147.986	6.2

We used a mesh containing 1002 nodes, which for a three-dimensional model is somewhat small. The accuracy (or lack of accuracy) of the above results is due to the coarseness of the mesh. If you use the GMSH graphical interface to examine this mesh you will notice that in the z direction there are something like 3 to 4 elements in the z direction. As a rule of thumb, we should try to get at least 10 elements in any direction if we need good accuracy. However, a model with a larger mesh will take much longer to execute on the computer. This problem was recalculated using a finer mesh. You can reproduce the finer mesh by making a change in the GEO file. At line (1) in the above GEO file listing change 1.7 to 0.9. This will produce a mesh having 5973 nodes. On a 900 Mhz pentium III PC with 512 Mb of RAM HMD took about 2 hours to solve this model. The resulting eigenvalues and the accuracy is shown in the table below.

eigenvalue (ω)	Theoretical	HMD output	deviation (%)
ω_{011}	112.128	113.349	1.1
ω_{111}	122.656	124.282	1.3
ω_{211}	135.250	137.434	1.6
ω_{021}	139.393	141.810	1.7

The output file, fe23.dat, is so large because it contains the eigenvectors. Each component of the vector is the scalar magnitude of the mode shape at the corresponding node location. Three-dimensional output is difficult to display without using powerful visualization software. But the HMD distribution comes with a utility program called INTP (interpolation program) which can scan the output file through a chosen plane to produce an image, like a CAT scan slice through the three-dimensional figure. With a simple model such as this we should be able to recognize radial and dimetral mode shapes in the interpolated slice.

To use INTP we need an input file with 5 columns. This is exactly the output format obtained when we run a POISSON or LAPLACE model with HMD. But the result of the eigenvalue model contains $N + 5$ columns. The HMD distribution comes with an AWK script, mode2datII.awk, for extracting a given eigenvector from the output file and producing a 5-column file usable by INTP. To extract the first mode vector from the output file use the following command:

```
cat fe23.msh | awk -f mode2datII.awk md=fe23.dat mdn=1 > mode1.dat
```

Now with mode1.dat we will use INTP to create the interpolated image slice. The program can produce its output either as an ASCII file readable by Matlab or as a binary image in raw (no header) format. The following command will produce a binary image file as output.

```
intp -d mode1.dat -g fe23.msh -b scalar -o fe23_1.raw -s -12, -12, 3, 12, -12, 3, 0, 12, 3
```

Notice that the -b argument specifies the output as binary. If you want an ASCII file instead, substitute a -a for the -b. The slice plane is defined by the -s argument. Think of the plane as defined by a square sheet that slices the figure. The square is defined by three points: two at its bottom corners and one at the midpoint of the top line. The -s argument takes 9 numbers which are the x, y, z coordinates of each of these three points. The first three numbers are the coordinates of the lower-left point. The second three numbers are the coordinates of the lower-right point, and the last three numbers are the coordinates of the upper, midsection point.

This program takes a long time to execute, as it is not a very efficient algorithm. When the program finishes we can convert the raw image format into a JPEG file or whatever other image format you like using Image Magick. The following command converts the raw image into a JPEG image.

```
convert -size 417x417 -depth 8 gray:fe23_1.raw fe23_1.jpg
```

Here is what the lowest mode shape, sliced through a plane bisecting the z axis, looks like in a 256 gray-level image. The model with 5973 nodes was used to produce this result.

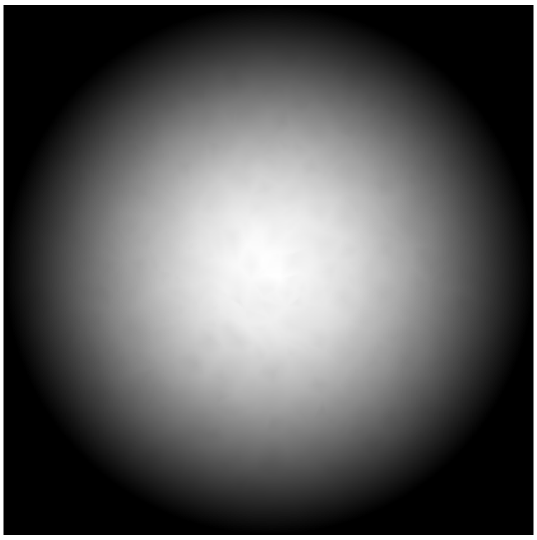


Figure 2. Mode 1

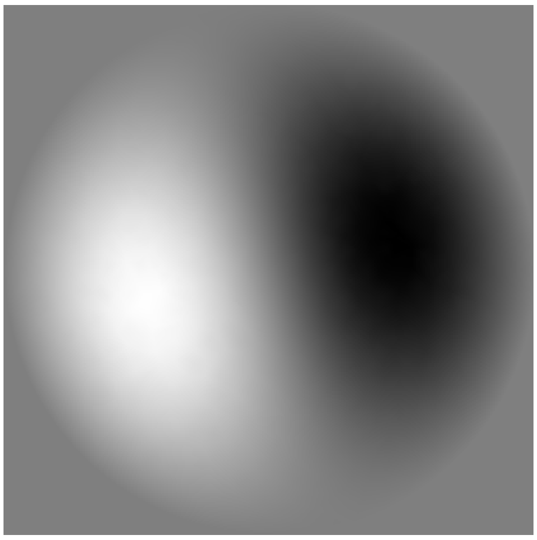


Figure 3. Mode 2

Chapter 7

Command Reference

§1. loadmesh.

description

Loads a mesh file previously set as the filename member of "mesh". The mesh variable name is the only argument, bcause the file name has already been set before this command is called. The mesh variable must have been declared before this command.

syntax

loadmesh <mesh_name>;

arguments

<mesh_name>

The variable name identifying the mesh. This is not a string, so don't enclose it in quotes.

§2. setregionbc.

description

Sets the desired *dirichtlet* boundary condition for the specified region.

syntax −− (mach I)

setregionbc <model_name> <region_number> <function_type> <constant> [(<axis> <order> <coefficient>), . . .] ;

syntax −− (mach II)

setregionbc <model_name> <region_number> <value or function_name>

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<region_number>

The region number. The boundary value will be applied to all nodes grouped into this region. The region numbers are given in the GMSH .geo and .msh files and must range from 1 to the maximum region number.

arguments −− (mach I only)

<function_type>

This is either "polynomial" or "rational" (in single quotes). The reason this is here is because the boundary value is ideally a function of x, y, and z. To set the boundary value to a constant, you just provide the zeroth-order term of a polynomial.

<constant>

The boundary value. This is actually the zeroth-order term of a polynomial, but if the boundary value is to be set as a constant this value will be the last parameter on the line.

optional mach I arguments

<axis>

If the boundary value is to be a polynomial function of x, y, and z, then this parameter identifies which direction. For x, x^2, x^3, … the parameter would be "x" (in single quotes). For any power of y the parameter would be "y", and so on.

<order>

The power of either x, y, or z.

<coefficient>

The polynomial coefficient of the term in x, y, or z.

arguments −− (mach II only)

<value or function_name>

Set a constant real value, or the name of a previously-defined function. If a function name, then it must have an underscore prefix. When you define this function, think of it like a callback function that expects to have 3 arguments passed to it: x, y, and z.

---

§3. setregionbcn.

---

description

Sets the desired *neumann* boundary condition for the specified region.

syntax −− (mach I)

setregionbcn < model_name>, < region_number>, < function_type>, < constant>, [(< axis>, < order>, < coefficient>), …] ;

syntax −− (mach II)

setregionbcn <model_name> <region_number> <value or function_name>

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<region_number>

The region number. The boundary value will be applied to all nodes grouped into this region. The region numbers are given in the GMSH .geo and .msh files and must range from 1 to the maximum region number.

arguments −− (*mach I only*)

<function_type>

This is either "polynomial" or "rational" (in single quotes). The reason this is here is because the boundary value is ideally a function of x, y, and z. To set the boundary value to a constant, you just provide the zeroth-order term of a polynomial.

<constant>

The boundary value. This is actually the zeroth-order term of a polynomial, but if the boundary value is to be set as a constant this value will be the last parameter on the line.

optional mach I arguments

<axis>

If the boundary value is to be a polynomial function of x, y, and z, then this parameter identifies which direction. For x, x^2, x^3, ... the parameter would be "x" (in single quotes). For any power of y the parameter would be "y", and so on.

<order>

The power of either x, y, or z.

<coefficient>

The polynomial coefficient of the term in x, y, or z.

arguments −− (*mach II only*)

<value or function_name>

Set a constant real value, or the name of a previously-defined function. If a function name, then it must have an underscore prefix. When you define this function, think of it like a callback function that expects to have 3 arguments passed to it: x, y, and z.

---

§4. setregiongrad.

description

Sets the stiffness coefficient for the model's GRAD (stiffness) matrix.

syntax

setregiongrad < model_name>, < region_number>, < constant_value> ;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<region_number>

The region number. The stiffness value will be applied to all GRAD elements grouped into this region. The region numbers are given in the GMSH .geo and .msh files and must range from 1 to the maximum region number.

<constant_value>

The value of the stiffness coefficient.

§5. setregionmass.

description

Sets the mass (inertial) coefficient for the model's MASS matrix.

syntax

setregionmass < model_name>, < region_number>, < constant_value> ;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<region_number>

The region number. The mass value will be applied to all MASS elements grouped into this region. The region numbers are given in the GMSH .geo and .msh files and must range from 1 to the maximum region number.

<constant_value>

The value of the inertial coefficient.

§6. setregiondamp.

---

description

Sets the damping (energy loss) coefficient for the model's DAMP matrix.

syntax

setregiondamp < model_name>, < region_number>, < constant_value> ;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<region_number>

The region number. The damping value will be applied to all DAMP elements grouped into this region. The region numbers are given in the GMSH .geo and .msh files and must range from 1 to the maximum region number.

<constant_value>

The value of the damping coefficient.

---

§7. setregionforce.

---

description

Sets the force (non-homogeneous term of the differential equation) coefficient for the model's vector.

syntax

setregionforce < model_name>, < region_number>, < constant_value> ;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<region_number>

The region number. The force value will be applied to all FORCE elements (element vectors) grouped into this region. The region numbers are given in the GMSH .geo and .msh files and must range from 1 to the maximum region number.

<constant_value>

The value of the force coefficient.

---

§8. buildglobalgrad.

description

This command performs the act of combining the individual element matrices into one global model matrix. In this case, the element matrices will be stiffness matrices. The resulting matrix will become a member data structure to the given model and be of matrix type **GRAD**.

syntax

buildglobalgrad < model_name>, < matrix_ID> ;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<matrix_ID>

The matrix ID is a string enclosed in single quotes. Here you are providing an identifier with which you may uniquely identify this matrix in the subsequent modeling commands.

§9. buildglobalmass.

description

This command performs the act of combining the individual element matrices into one global model matrix. In this case, the element matrices will be mass matrices. The resulting matrix will become a member data structure to the given model and be of matrix type **MASS**.

syntax

buildglobalmass < model_name>, < matrix_ID> ;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<matrix_ID>

The matrix ID is a string enclosed in single quotes. Here you are providing an identifier with which you may uniquely identify this matrix in the subsequent modeling commands.

§10. buildglobaldamp.

description

This command performs the act of combining the individual element matrices into one global model matrix. In this case, the element matrices will be damping matrices. The resulting matrix will become a member data structure to the given model and be of matrix type **DAMP**.

syntax

buildglobaldamp < model_name>, < matrix_ID>, < detID or flags > ;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<matrix_ID>

The matrix ID is a string enclosed in single quotes. Here you are providing an identifier with which you may uniquely identify this matrix in the subsequent modeling commands.

< detID >

This optional 3rd argument gives the ID string of a determinant struct (the object that is built by createsparsity), that already exists, from which this matrix will derive its properties. If no 3rd argument is given, then the first determinant struct belonging to the model is used (the sparsity map created by the first call to createsparsity or when the first model matrix was built).

or

< flags >

If the optional 3rd argument is an integer instead of "detID", then this is understood to be a set of bit-coded flags having the same meanings as the flags given to the createsparsity command. If flags are given to this command, then a new determinant struct (sparsity map) is created just as if the createsparsity command were being called. If no 3rd argument is given, then the first determinant struct belonging to the model is used (the sparsity map created by the first call to createsparsity or when the first model matrix was built).

§11. buildglobalvector.

description

This command performs the act of combining the individual element vectors into one global model vector. The resulting vector will become a member data structure to the given model and be of vector type **FORCE**.

syntax

buildglobalforce < model_name>, < vector_ID> ;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<vector_ID>

The vector ID is a string enclosed in single quotes. Here you are providing an identifier with which you may uniquely identify this vector in the subsequent modeling commands.

---

§12. setdivergence.

---

description

Sets the matrix type for a model matrix to be of type DIVERGENCE. This matrix type designates that it is ready to have boundary conditions set and to be solved.

For LAPLACE's equation, the matrix going into solution activities should be of type DIVERGENCE.

For POISSON's equation, solution activities expect a matrix of type DIVERGENCE and a vector of type SOURCE (see setsource).

For the EIGENMODE equation, solution activities expect a stiffness matrix set to type DIVERGENCE and a mass matrix set to type EIGENMATRIX (see seteigenmatrix). If no matrix of type EIGENMATRIX is found the solution activities assume that the mass matrix was inverted and multiplied into the stiffness matrix, so that a "standard eigenvalue" problem is assumed.

syntax

setdivergence < model_name>, < matrixID>;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<matrixID>

The matrix ID string (enclosed in single quotes) of the matrix whose type will be changed to DIVERGENCE.

---

§13. seteigenmatrix.

---

description

Sets the matrix type for a model matrix to be of type EIGENMATRIX. This matrix type designates that it is ready to have boundary conditions set and to be solved.

For LAPLACE's equation, the matrix going into solution activities should be of type DIVERGENCE.

For POISSON's equation, solution activities expect a matrix of type DIVERGENCE and a vector of type SOURCE (see setsource).

For the EIGENMODE equation, solution activities expect a stiffness matrix set to type DIVERGENCE and a mass matrix set to type EIGENMATRIX (see seteigenmatrix). If no matrix of type EIGENMATRIX is found the solution activities assume that the mass matrix was inverted and multiplied into the stiffness matrix, so that a "standard eigenvalue" problem is assumed.

syntax

seteigenmatrix < model_name>, < matrixID>;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<matrixID>

The matrix ID string (enclosed in single quotes) of the matrix whose type will be changed to EIGENMATRIX.

---

§14. setsource.

---

description

Sets the vector type for a model vector to be of type SOURCE. This vector type designates that it is ready to have boundary conditions set and to be solved.

For LAPLACE's equation, the matrix going into solution activities should be of type DIVERGENCE.

For POISSON's equation, solution activities expect a matrix of type DIVERGENCE and a vector of type SOURCE (see setsource).

For the EIGENMODE equation, all vectors are ignored.

syntax

setsource < model_name>, < vectorID>;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<vectorID>

The vector ID string (enclosed in single quotes) of the vector whose type will be changed to SOURCE.

---

§15. combinematrices.

description

Adds two matrices belonging to the same model. The resulting matrix replaces the first matrix.

syntax

combinematrices < model_name>, < matrix_1>, < matrix_2> ;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<matrix_1>

The matrix ID string (enclosed in single quotes) of first matrix to combine. This matrix will be overwritten by the resulting matrix.

<matrix_2>

The matrix ID string of the second matrix to combine (enclosed in single quotes).

§16. combinemodels.

description

Mach I only. Combines several individual models into one supermodel in which each of the component matrices form blocks in the resulting block matrix. The sequence of arguments lists the models and their corresponding matrix ID strings as block positions, from left to right and top to bottom order, in the block matrix. After the block dimension is specified, the arguments follow in triplets of **model**, **matrix**, and **vector**, until every block in the final matrix has been provided.

syntax

combinemodels < supermodel_name>, < block_dim>, < model_1>, < mID_1>, < vID_1>, . . . ;

arguments

<supermodel_name>

The variable name identifying the final model given as a string enclosed in single quotes.

<block_dim>

The matrix dimension, in blocks, of the final matrix.

<model_1>

The first model name given as a string enclosed in single quotes.

<mID_1>

The first matrix ID string enclosed in single quotes.

<vID_1>

The first vector ID string enclosed in single quotes.

---

§17. insertbcforce.

---

description

Mach I only. Inserts the boundary values into the model vector. This command must be called before the corresponding command for the model matrix. This is because for dirichtlet boundary conditions the boundary value must be first multiplied on each matrix value in the corresponding column and added to the vector. The process of inserting dirichtlet boundary values into the matrix involves setting the corresponding row and column values to zero, so the vector boundary operations must be done before the matrix column is set to zero.

syntax

insertbcforce < model_name> ;

arguments

<model_name>

The variable name identifying the model given as a variable (not in quotes).

---

§18. insertbcmatrix.

---

description

Mach I only. Inserts the boundary values into the model matrix.

syntax

insertbcmatrix < model_name> ;

arguments

<model_name>

The variable name identifying the model given as a variable (not in quotes).

---

§19. insertbcmodel.

description

Mach II only. Inserts the boundary values into the model matrix and vector. If no matrix and vector arguments are given, then it finds the first matrix belonging to this model of type DIVERGENCE and the first vector belonging to this model of type SOURCE.

syntax

insertbcmodel < model_name> ;

insertbcmodel < model_name>, < matrix_ID>, < vector_ID> ;

arguments

<model_name>

The variable name identifying the model given as a variable (not in quotes).

§20. SOLVE.

description

Mach I only. Solves the matrix equation representing the finite element model.

syntax

SOLVE < model_name> ;

arguments

<model_name>

The variable name identifying the model given as a variable (not in quotes).

§21. vec2file.

description

Mach I only. Saves the solution vector to an ascii file as a set of five columns: the node number, the node value, the node x position, the y position, and the node z position.

syntax

vec2file < model_name>, < file_name> ;

arguments

<model_name>

The variable name identifying the model given as a string enclosed in single quotes.

<file_name>

The desired name of the output file given as a string enclosed in single quotes.

---

§22. solve_div.

---

description

Mach II only. The equivalent of SOLVE but for Mach II models.

syntax

solve_div < model_name>, < output_filename>, <flags>, < node-specifiers> ;

arguments

<model_name>

The name of the model to be solved.

<output_filename>

The name of the output file to be created.

<flags>

A bit-coded interger (decimal or hexidecimal). A zero will work.

bit 5	0 = unpacked storage, 1 = band packed storage (SPARSPAK)

<node-specifiers>

The remaining arguments in the list specify nodes, or ranges of nodes. You may just put 'all' and compute all the nodes. Or you may specify individual nodes like, 1, 2, 3, 5, 8, etc. Or you may specify ranges of nodes like 1:7, 8:12, 20:25, etc.

---

§23. solve_eigen.

---

description

Mach II only. The equivalent of SOLVE but for Mach II models.

syntax

solve_eigen <model_name>, <output_filename>, <flags>, <node-specifiers> ;

arguments

<model_name>

The name of the model to be solved.

<output_filename>

The name of the output file to be created.

<flags>

A bit-coded interger (decimal or hexidecimal). A zero will work.

bit 0	must be zero
bit 1	0 = symmetric matrix, 1 = nonsymmetric matrix
bit 2	0 = leave out unit vector space, 1 = total matrix
bit 3	0 = generalized eigenvalue, 1 = standard
bit 4	0 = unpacked storage, 1 = symmetric packed storage
bit 5	0 = unpacked storage, 1 = band packed storage (SPARSPAK)
bit 7	0 = calculate eigenvectors also, 1 = eigenvalues only

A flag of 0x00 will work most often. For packed storge use 0x10. For packed storage and to leave out the eigenvectors, use 0x90.

<node-specifiers>

The remaining arguments in the list specify nodes, or ranges of nodes. You may just put 'all' and compute all the nodes. Or you may specify individual nodes like, 1, 2, 3, 5, 8, etc. Or you may specify ranges of nodes like 1:7, 8:12, 20:25, etc.

The output file contains the eigenvalues and the eigenvectors. For all models the first 5 columns of the file are the node number, eigenvalues, and x, y, z coordinates, respectively. The remaining matrix of numbers to the right of the coordinates will be the eigenvectors.

You should expect the columns (starting with column 6) to be eigenvectors. The vectors will be in the same order as the eigenvalues. The eigenvalues corresponding to the boundary contraints come out as unity, and the corresponding eigenvector comes out as a unit vector. The ordering of the eigenvalues appears arbitrary–the unity eigenvalues are found all gathered at the end of the list even though they correspond in some way to the boundary nodes–so don't expect to match eigenvalues (and their vectors) to the node numbers in column 1. But the components of the vectors should match the node numbers in column 1, which is why I print the vectors as columns in the output file.

§24. setexpandelements.

---

description

Mach II only. A debugging tool. You can view the individual elements matrices created for the mesh before they are added into the global matrix. This command sets a global flag that is private to the HMD code which will save the element matrices to the ascii file specified in the command.

syntax

setexpandelements <model_name>, <mesh_name>, <output_filename> ;

arguments

<model_name>

The name of the model for which the elements are created, enclosed in single quotes.

<mesh_name>

The name of the mesh for which the elements are created, enclosed in single quotes.

<output_filename>

The name of the output file to be created, enclosed in single quotes.

---

§25. expandmatrix.

---

description

Mach II only. Output a matrix to an ascii text file readable by Matlab and other common tools. The matrix can be output at any stage of the model. For example, you can save the matrix immediately after it was assembled from the element matrices, or after the boundary values have been inserted.

Each line of the file is a row of the matrix. The matrix elements are separated by spaces. To read the matrix into Matlab, use the dlmread command. To read it into Scilab, use the read command.

syntax

expandmatrix <model_name>, <matrix_ID>, <output_filename> ;

arguments

<model_name>

The name of the model to which this matrix belongs, enclosed in single quotes.

<matrix_ID>

The name of the matrix, or ID string, assigned to this matrix, enclosed in single quotes.

<output_filename>

The name of the output file to be created, enclosed in single quotes.

§26. exportmatrix.

description

Same as **expandmatrix**, but for Mach I models.

§27. setbandpacked.

description

Mach I POISSON only. Reorder the sparse, symmetric matrix into a band structure using the SPARSPAK routines. Call this command after assigning the boundary values to the region numbers (with setregionbc) but before the global matrices are assembled. The solver knows to look for the modified band structure in the matrix and loads it appropriately into the LAPACK matrix solver.

syntax

setbandpacked <model_name> ;

arguments

<model_name>

The name of the model to which this matrix belongs without quotes.

§28. createsparsity.

description

Mach II only. Builds a sparsity map of the matrix equation so that only the non-zero matrix elements are stored. The storage format is not like the Matlab row,column,value triplets but is a special format. The matrices and vectors are stored, sparsley, either in memory or on disk (that's right, not in memory) depending on the bit flags that are given to this command. Also, this command sets the precision, single or double, with the bit flags.

Note that this command MUST be used with Mach II models.

syntax

createsparsity <model_name> , < flags > ;

arguments

<model_name>

The name of the model enclosed in single quotes.

<flags>

A bit-coded interger (decimal or hexidecimal). A zero will work.

bit 0	0 = save the sparsity table in memory, 1 = save to file
bit 1	0 = matrices in memory, 1 = matrix to file
bit 2	0 = vectors in memory , 1 = vectors to file
bit 5	0 = single precision, 1 = double precision
bit 6	0 = real-valued, 1 = complex
bit 7	1 = modal matrix (special damped eigenmatrix)
bit 8	0 = symmetric matrix, 1 = nonsymmetric

For example, 0x00 means single precision, 0x20 means double precision. For single precision and file storage for the sparsity table use 0x01.

description	flags (hex)
All data in memory, single precision, real	0x00
All data in memory, double precision, real	0x20
All data in memory, single precision, complex	0x40
All data in memory, double precision, complex	0x60
Sparsity table in file, single precision, real	0x01
Sparsity table in file, double precision, real	0x21
Sparsity table in file, single precision, complex	0x41
Sparsity table in file, double precision, complex	0x61
Matrices in file, single precision, real	0x02
Matrices in file, double precision, real	0x22
Matrices in file, single precision, complex	0x42
Matrices in file, double precision, complex	0x62
Vectors in file, single precision, real	0x04
Vectors in file, double precision, real	0x24
Vectors in file, single precision, complex	0x44
Vectors in file, double precision, complex	0x64
All data in memory, single precision, real, modal	0x80

§29. set_region_force_priorities.

description

Since force values are ultimately assigned to global nodes, but any one node may be owned by more than one element, which belong to different regions, some nodes may not get values set that we expect.

This function assigns a priority to the regions for inserting force (load) values in decreasing order: the first number in the list has the highest priority. If fewer region numbers are given than the total regions, this function completes the priority list with the remaining region numbers in arbitrary order.

For example, node 35 might be owned by element 115, which belongs to region 1, and element 121, which belongs to region 7. To insure that region 7 force values take priority over all others put 7 as the first number in the priority list.

syntax

set_region_force_priorities <model_name> , < r1, r2, . . . > ;

arguments

<model_name>

The name of the model enclosed in single quotes.

< r1, r2, . . . >

The region numbers given as highest priority first. The complete set of region numbers is not required to be given.

---

§30. buildglobal_invmatrix.

---

description

Mach II only. Builds an inverse matrix, stored in mach II sparsity format of an existing model matrix.

syntax

buildglobal_invmatrix <model_name> , < flags >, < source_matrix_ID >, < destination_matrix_ID > ;

arguments

<model_name>

The name of the model enclosed in single quotes.

<flags>

A bit-coded interger (decimal or hexidecimal). A zero will work.

bit 0	0 = the source matrix is symmetric (typical), 1 = the source matrix is nonsymmetric

For example, 0x00 means symmetric, 0x01 means nonsymmetric.

< source_matrix_ID >

The matrix ID string (enclosed in single quotes) of the existing model matrix to be inverted.

< destination_matrix_ID >

The matrix ID string (enclosed in single quotes) which will be assigned to the inverted matrix.

---

§31. buildglobal_matrixproduct.

description

Mach II only. Builds a matrix, stored in mach II sparsity format, that is the matrix product of two existing model matrices.

syntax

buildglobal_matrixproduct <model_name> , < flags >, < 1st matrix_ID >, < 2nd matrix_ID > < product matrix_ID >;

arguments

<model_name>

The name of the model enclosed in single quotes.

<flags>

A bit-coded interger (decimal or hexidecimal). A zero will work.

< 1st matrix_ID >

The matrix ID string (enclosed in single quotes) of an existing model matrix.

< 2nd matrix_ID >

The matrix ID string (enclosed in single quotes) of an existing model matrix.

< product matrix_ID >

The matrix ID string (enclosed in single quotes) that will be assigned to the product matrix.

§32. impandmatrix.

description

Mach II only. Imports a matrix from an ascii file in Matlab matrix format and converts it into the Mach II sparsity structure. The contrived name "impandmatrix" is meant to be antisymmetric to the command "expandmatrix".

syntax

impandmatrix <model_name> , < flags >, < matrix_ID >, < file name > < dimension >;

arguments

<model_name>

The name of the model enclosed in single quotes.

<flags>

A bit-coded interger (decimal or hexidecimal). A zero will work.

bit 0	0 = the source matrix is symmetric (typical), 1 = the source matrix is nonsymmetric

< matrix_ID >

The matrix ID string (enclosed in single quotes) to be assigned to the matrix after it is loaded.

< file name >

The name of the ascii file containing the matrix.

< dimension >

The dimension of the matrix. The matrix is expected to be square.

---

§33. multiply_by_constant.

---

description

Mach II only. Multiplies a given model matrix by a constant. The matrix may be real or complex, and the constant may be real or complex.

syntax

multiply_by_constant <model_name> , < matrix_ID >, < real_part >, < imag_part >;

arguments

<model_name>

The name of the model enclosed in single quotes.

< matrix_ID >

The matrix ID string (enclosed in single quotes) of an existing model matrix.

< real_part >

The real part of the constant. If the constant is real, then this is just the constant.

< imag_part >

The (optional) imaginary part of the constant. If the constant is real, then this argument will be absent.

---

§34. convert_to_complex.

---

description

Mach II only. Converts a real model matrix to complex;

syntax

convert_to_complex <model_name> , < matrix_ID >;

arguments

<model_name>

The name of the model enclosed in single quotes.

< matrix_ID >

The matrix ID string (enclosed in single quotes) of an existing model matrix.

(in progress)

Chapter 8

Utilities

1. INTP

2. DAT2VIS5D

§1. INTP.

INTP is a utility for interpolation. It is used for creating a regularly-spaced array of interpolated output values from the node-value data produced by HMD (or any other finite element program). Recall that the solution output from HMD is a file with 5 columns: the node number, the value at that node, and the x, y, and z coordinates of the node. These node positions are arbitrarily distributed throughout the mesh. This kind of data format is not easily displayed with typical plotting programs. It can be done with Matlab, but the procedure is not straightforward.

INTP needs the output file (5-column format) from HMD and the GMSH file (.msh format) that was used by HMD. Using these two files as input, INTP will interpolate the values between the nodes to produce an M columns by N rows data file, either ascii or binary. You can get the scalar data values or get the x derivatives or get the y derivatives.

INTP can interpolate the output from either a two-dimensional model or a three-dimensional model. But the output from INTP is always an array of rows and columns, that is, an image of the data. So if you interpolate a 3D model you must select a slice plane through the model. There are two ways to take a slice through a 3D model: one way is slow and the other is fast. However, the output usually looks the same, so you should try the fast way first.

2-D Models

The basic usage of INTP is to specify the data file, the mesh file, ascii or binary, and the output file name. Let's say you have an HMD model consisting of my_model.geo, my_model.msh, my_model.hmd, and you created an HMD output file called my_model.dat. The following example shows how to create a file readable with Scilab or Matlab as a 2D array (matrix) of ascii values called matlab.dat.

```
intp -d my_model.dat -g my_model.msh -a scalar -o matlab.dat
```

Notice the **-a** option followed by **scalar**. This is what distinguishes the output type between ascii and binary. In order to create an output file of biary numbers we would substitute the **-a** option with **-b**. The following example is the same as the one above but produces raw, 8-bit binary data.

```
intp -d my_model.dat -g my_model.msh -b scalar -o image.raw
```

The output file is **image.raw**. To display this data you can read it into Matlab or convert it into a standard image format. You can use ImageMagick (freely available) to do the conversion. The following ImageMagick command will convert the binary data into a JPEG file.

```
convert -size 417x417 -depth 8 gray:image.raw image.jpg
```

Note that the default size of the data array is usually, but not always, 417 by 417. INTP will display the data size of its output on the console and will automatically create a file called intpsize.dat that contains the image size.

INTP will also create a binary color image in RGB format. Each pixel is represented by a sequence of three 8-bit intergers in red, green, and blue order. The following INTP command will produce a color binary image:

```
intp -d my_model.dat -g my_model.msh -B scalar -o image.raw
```

Notice that the only difference between the command line for color is that the -b argument becomes upper case for color. The following ImageMagick command will convert the RGB binary data into a color JPEG file.

```
convert -size 417x417 rgb:image.rgb image.jpg
```

3-D Models

Using INTP on a 3D model is the same as for a 2D model, with the exception that you must also specify a slice plane. The slice plane is given by three points, but these points cannot be any arbitrary points. Point 1 must be the lower-left corner; point 2 must be the lower-right cormer; point 3 must be the mid-point on the top edge of the plane. For example, let's say the model is defined within a cube centered at the origin with side of 10, 10, and 10. Let's interpolate a slice through the x,z plane at $y = 0$. Then you must define point 1, $P_1 = (x_1, y_1, z_1)$, which will have coordinates $x_1 = -5, y_1 = 0, z_1 = -5$. Point 2, $P_2 = (x_2, y_2, z_2)$, will have coordinates $x_2 = 5, y_2 = 0, z_2 = -5$. Point 3, $P_3 = (x_3, y_3, z_3)$, will have coordinates $x_3 = 0, y_3 = 0, z_3 = 5$. The slice plane coordinates are given with the **-s** option according to the following syntax:

```
-s      x1, y1, z1, x2, y2, z2, x3, y3, z3
```

Please do not put spaces between the coordinate numbers and the commas. Now the complete command syntax for interpolating a slice through the 3D model is

```
intp -d my_model.dat -g my_model.msh -a scalar -o matlab.dat
-s -5,0,-5,5,0,-5,0,0,5
```

Fast 3-D Interpolation

The 3D slice can be interpolated very much faster by creating an INTP "cut mesh". INTP will create a secondary, 2D, mesh defined by the intersections of the 3D mesh with the slice plane. This cut mesh can be feed back into INTP as if it were a standard 2D mesh. The **-k** option produces a cut mesh and created a data file called cutmesh.dat and a mesh file called cutmesh.msh. The following command will produce a cut mesh from the 3D model as above.

```
intp -d my_model.dat -g my_model.msh -k 1 -s -5,0,-5,5,0,-5,0,0,5
```

§2. DAT2VIS5D.

DAT2VIS5D is a utility for stacking and converting image slice data into **vis5d+** format. **vis5d+** is a free-available visualization program for 3D display and animation (http://vis5d.sourceforge.net). Unfortunately, the input file format for **vis5d+** is binary, so a program is needed just for converting the input data. However, it is a nice package for display of atmospheric and oceanographic physics.

Output data from HMD must first be interpolated using INTP into image slices. Each image slice will be a separate binary file. DAT2VIS5D will convert all the image slices into one binary file that is read by **vis5d+**. The DAT2VIS5D program needs a command file to tell it how to combine the data. There is an example command file in the HMD examples directory called "v5dconfig.cfg".

Once you have created your command file for DAT2VIS5D using a text editor, you must invoke it using the following command:

dat2vis5d -f command_filename -o output_filename

The resulting the file will be an input file for **vis5d+**.

(in progress)

§3. WAVY.

WAVY is a utility for creating a time simulation from an HMD eigenvalue (vibrational modes) model. Given a shape function, that is, the initial condition as a set of position coordinates and amplitudes, the harmonic contribution from each normal mode required to describe this shape is calculated. The time solution is then an expansion in normal modes where each mode is multiplied by its strength (shape coefficient) and $\cos(\omega_n t)$. The following example shows how to generate a sequence of time-frame images:

```
wavy -r 101 -s shapefile -d my_model.dat -n numnodes -T 0.001 -N 100
```

The above command takes the HMD output file my_model.dat and a single-column file that describes the initial shape. The number of nodes is taken to be the number of rows that appear in the HMD output file. Each frame will be separated in time by 0.001 seconds, and there will be 100 frames. The files that are created will automatically be called f1.dat, f2.dat, etc.

After creating the images they may be assembled into an MPEG movie using mpeg_encode (the Berkeley encoder). I have found that the easiest way to create a movie with mpeg_encode is to convert the images into PNM format with ImageMagick. The following script will create a basic movie for you:

```
PATTERN I
INPUT_DIR .
INPUT
f*.pnm [1-100]
END_INPUT
OUTPUT mpeg_movie.mpg
GOP_SIZE 100
INPUT_CONVERT *
BASE_FILE_FORMAT PNM
SLICES_PER_FRAME 1
PIXEL HALF
PSEARCH_ALG LOGARITHMIC
BSEARCH_ALG SIMPLE
IQSCALE 8
PQSCALE 10
BQSCALE 25
REFERENCE_FRAME ORIGINAL
RANGE 10
FRAME_RATE 30
```

(in progress)

Chapter 9

Limitations and Future Development

§1. Known Bugs.

§2. Mesh File Node Numbering.

It will often happen that the mesh file *file.msh* resulting from GMSH has a node list where some of the node numbers are missing. This is not a bug in GMSH. GMSH was not designed to produce node numbers in order, and strictly speaking the finite element method does not require this.

However, HMD needs to have the nodes numbered from 1 to N. If not, the results will be unpredictable–your model will not work. Always check the .msh file. If the node numbering is wrong you can fix it with an AWK script I have provided called reorder.awk:

awk -f reorder.awk *mesh_file* > *temporary_file*

§3. Cell Grid Shaping.

At present the cell grid can be created only as a rectangular, N-dimensional cube. Future development will include commands for creating other geometric shapes such as circles, spheres, toriods, etc.