School of Engineering
Brown University

EN234: Computational methods in Structural and Solid Mechanics

EN234FEA TUTORIAL

BEFORE DOING THIS TUTORIAL YOU WILL NEED TO INSTALL FORTRAN, TECPLOT, AND ECLIPSE, SET UP A GITHUB ACCOUNT, FORK THE EN234FEA GITHUB REPOSITORY, AND CLONE THE PROJECT INTO ECLIPSE. Follow the instructions posted at http://www.brown.edu/Departments/Engineering/Courses/En2340/Programming/programming.html**.**

Overview

EN234FEA provides a basic set of subroutines that take care of most of the data management associated with running a finite element analysis. It is intended to provide a platform for developing and testing new element types or new materials; or to develop finite element methods that might require access to variables (such as global stiffness data) that can be difficult to access in commercial codes. Specifically, the code has capabilities to

- Read and parse an ascii input file that defines the mesh, loading conditions, type of analysis and various post-processing options
- Assemble the global finite element stiffness matrix and global stiffness matrix given the corresponding element matrices
- Solve the system of equations. Procedures are included for (i) a linear problem; (ii) the nonlinear Newton-Raphson method and (iii) explicit dynamics. A direct solver and nonsymmetric conjugate gradient are provided.
- Handle basic adaptive time stepping for nonlinear problems.
- Handle various boundary conditions that arise in finite element analysis, such as prescribed displacement histories; prescribed tractions, etc
- A least-squared fitting procedure to determine nodal values of element quantities that are usually defined at integration points. This generally improves the accuracy of these quantities (and so provides a simple error estimate, eg for adaptive meshing); and is needed to be able to plot contours in TECPLOT
- Print variables (eg the mesh, nodal degree of freedom values, and projected nodal values of element quantities) to files that can be read by TECPLOT.

Several routines are provided with the intention that users will modify them

- A stub for a new element
- A stub to enable the user to print variables of interest to file(s) during an analysis
- Stubs to define user-controlled constraints and boundary conditions.

In addition, various routine element operations for continuum elements have been pre-defined, including

- Integration points for most standard 1D, 2D and 3D solid elements
- Shape functions and their derivatives for standard linear and quadratic triangular; quadrilateral; tetrahedral; and hexahedral elements.
- Utilities for some routine operations often needed in element or material-level FEA simulations for solids, such as inverting 2x2 or 3x3 matrices; computing eigenvalues/eigenvectors of symmetric two-tensors; computing the square root of a symmetric tensor, and so on.

As an example, element routines are provided for a standard fully integrated 3D linear elastic element.

1. **Editing EN234FEA and running your modified code**

   - Start by selecting the input file you will run. To do this, open the EN234_FEA project in Eclipse, then expand the src directory in the Project Explorer window, and double-click main.f90 to open it. Edit the lines of code that specify the input and output file to use ./input_files/Holeplate_3d.in for the input file; and ./output_files/Holeplate_3d.out for the output file. **SAVE THE edited main.f90 FILE (if you make a change and your code does nothing new it is probably because you forgot to save your edits!)**
   - Compile the modified code. Select Project > Build All (If you have more than one Eclipse Project you may need to select Build Project instead)
   - Select the executable in the Binary directory and select Run
   - Right click the Output Files directory in the Project Explorer and select Refresh
   - Open the Holeplate_3d.out file and check that the code ran correctly

2. **Committing your changes to the code and pushing them to your remote repository.**

   - Select Git > Commit (or use the commit button). A menu will come up asking you to enter a description of your change – enter a short description such as 'edited main.f90 to read Holeplate_3d.in'. Below that it will ask you which files you wish to commit. It is not worth version tracking output files for this class (although you may wish to do so when archiving your research results) so just check the box for main.f90. Then select Commit. This has recorded your changes on your *local* computer. It is not yet recorded in your online GitHub repository.
   - Select Git > Push to Upstream (or use the push button). You will see a description of the changes that will be pushed to your remote repo. Click
   - Go to your online GitHub page and find your fork of the EN234_FEA repository (it should be listed under your repository). Refresh the page if necessary. You should see that your changes have been committed.

3. **Using TECPLOT to visualize the results.**

   - To view the results, open tecplot (I suggest using Tecplot Focus), then select File > Load Data File > and select the Tecplot Data Loader. You will have to find the path of the Output files directory – you can do this by finding the output_files/contourplots.dat file in the Project Explorer window of Eclipse, right clicking the file, and select Properties. You can copy and paste the Location into the Tecplot window. Accept the default in the next menu to plot a 3D view of the object. To display the mesh or contours click the radio buttons on the left. You can select what variable to plot in the contours using the […] button to the right of the contour checkbox. The input file for EN234_FEA controls how the variables are labeled.

Play with TECPLOT for a while to get comfortable with the various buttons. The environment is fairly self-explanatory. Try and see if you can work out how to do all of the following:

- Changing the view
- Removing or modifying the axes
- Moving or rescaling the contour legend
- Removing the header, border or background from the window
- Add some text strings
- Change the color of the mesh
- Make an animated sweep showing the through-thickness variation of one of the contour variables

TECPLOT is a very nice program – it is much easier to use, and produces much nicer output that ABAQUS/CAE. Actually TECPLOT 360 will read an ABAQUS .odb file and you might like to explore this capability even for plotting ABAQUS results.

4. Interpreting the input file

Open the example file Holeplate_3d.in. Here are some general instructions on modifying the file:

- The % symbol is a comment marker and anything following a % is ignored by the code.
- **Data Lines cannot exceed 100 characters in length.** (comments can be as long as you like)
- The input file has 3 major sections (i) mesh definition (which includes defining material properties); (ii) boundary condition definition; and (iii) Analysis.

Mesh Definition:

You can create a mesh in one of two ways: (1) Read the mesh from the input file; and (2) write code to create the mesh directly.

To read the mesh from a file; you must define the nodes and their properties (coordinates, number of degrees of freedom, etc); and define the elements (connectivity; material properties; and any material state variables)

NODAL PROPERTIES

- The NODES key starts definition of node properties. The END NODES key terminates it.
- The PARAMETERS key specifies properties of the nodes: you must specify the number of coordinates; the number of unknowns (degrees of freedom) and you can provide an optional integer number that will help you distinguish between different types of node in an analysis.
- Then, following the COORDINATES keyword, you must enter the coordinates for each node. The node numbering scheme for the standard 2D and 3D element shape functions provided in EN234FEA are described in chapter 8 of http://solidmechanics.org (and are the same as ABAQUS, so you can cut and paste an ABAQUS mesh out of an ABAQUS .inp file into an EN234FEA input file. You must end coordinate definition with an END COORDINATES
- The DISPLACEMENT DOF key allows you to specify which degrees of freedom for your node represent displacements. These are needed to plot a displaced mesh correctly in a multi-physics analysis. By default, the first three DOF are assumed to be displacements in the three coordinate directions.
- You can create nodes with different properties in the same analysis – in a multi-physics problem it is common to include nodes that have different numbers of types of degrees of freedom. To do this simply repeat the PARAMETERS and COORDINATES blocks for each group of nodes.

ELEMENT PROPERTIES

- The ELEMENT key begins element definition. An ELEMENTS block must be terminated by an END ELEMENTS
- The PARAMETERS key specifies the nature of the elements. You must supply the no. nodes on the element, the number of history dependent element state variables (eg plastic strain values at integration points), and an integer identifier that will enable you to distinguish between different element types during an analysis.
- Following the PROPERTIES key you must enter values for the element properties (eg material properties). Note the 100.d0 – this is FORTRAN notation for a double precision number 100 x 10^0. It is not necessary to use FORTRAN notation – the code will interpret anything that looks like a

number. All the elements listed in the subsequent CONNECTIVITY key will have the same properties.

- For an explicit dynamic computation you must provide a DENSITY key followed by a value for the density.
- Then, following the CONNECTIVITY keyword you must specify the node numbers that lie on the vertices of each element in the mesh.
- All the elements defined in this way will have the same properties. If you want to solve problems with inhomogeneous materials you can enter another ELEMENT keyword and repeat the procedure. Note that the code assumes that element state variables (but not degrees of freedom) are discontinuous across boundaries between different regions when the element state is projected to determine nodal values of the state variables (new nodes are introduced automatically so the discontinuities are plotted correctly by TECPLOT)

You can also write code to create a mesh. For this purpose you must (i) enter the following keys in the input file

```
MESH
  USER SUBROUTINE
    Specify a list of integer or real valued parameters that control the nature of your mesh
  END USER SUBROUTINE
END MESH
```

and (ii) edit the source code in the file called user_mesh.f90 to define your mesh. Comments in the file will tell you which variables must be allocated and/or initialized in the file.

Boundary condition Definition: has the following sections:

- Defining load or displacement histories. You might want to make the displacement of one or more nodes vary with time in some way (eg ramp up the displacement), or make the tractions applied to an element face vary with time. In this case you need to define the history. The HISTORY keyword lets you do this. Enter HISTORY, and then assign a name (a character string) to the history so you can refer to it later. Then enter pairs of values of (time, value) to define the history. The code will interpolate linearly with respect to time to determine the value of the load or displacement. If during the analysis the time exceeds the last time value provided in a table, the load (or displacement) will be kept constant.
- You can also specify the time history of degrees of freedom or tractions through a user-subroutine. In this case you can enter a list of parameters that control the behavior of the user subroutine using a USER SUBROUTINE PARAMETERS block. Enter USER SUBROUTINE PARAMETERS followed by a name, then on the following lines enter a list of integers or real numbers. End the block with an END SUBROUTINE PARAMETERS.
- Defining groups of nodes and elements. The NODE SET defines a group of nodes, the ELEMENT SET defines a group of elements. Note that unlike ABAQUS you can't create arbitrary numbers for nodes and elements. The node numbers and element numbers are determined by the order in which they are defined in your input file (the first node for which you specify coordinates is node number 1, and the first element for which you specify a connectivity is element number 1, for example).
- Assigning values or time histories to degrees of freedom. Use the DEGREE OF FREEDOM block to do this. The conventions are hopefully self-explanatory. For example the line node1, 3, VALUE, 0.0 assigns the value zero to U3 to the 3rd node in the mesh (because the node set node1 contains only node 3). The right, 1, dof_history assigns the time variation specified in the history table called dof_history to the nodeset called 'right.'

- You can also enter distributed loads acting on element faces (the face numbering convention is listed in chapter 8 of http://solidmechanics.org)
- In a static analysis, you can define constraints between degrees of freedom.

Analysis Definition:

EN234FEA can run two different types of analysis: (1) a static linear or nonlinear analysis; (2) an explicit dynamic analysis. The Holeplate_3d.in file uses a static analysis. The following parameters must be specified in this case:

- Time step definition. You must enter an initial value for the time increment in the analysis; the maximum and minimum allowable values (the code will try to take the maximum allowable value, but if convergence fails will reduce the time step. If the times step falls below the minimum value it will terminate with an error). Then you must enter the maximum number of time steps. Finally you must specify the time increment and number of steps between printing the datafile with the mesh and contour data (the code will use whichever results in the larger number of files); and the same data for user-defined printing (the large values in the example file mean the user-defined printing is never activated).
- Specify the solution method. The options are specified in the input file as comments – you can run a linear analysis (no Newton-Raphson iterations are used); or a nonlinear analysis (which will use Newton-Raphson iteration – in this case you must specify a value for the convergence tolerance (typically 0.0001); and a maximum number of allowable iterations (eg 15). If the maximum iterations are exceeded the code will reduce the time increment and attempt to find the solution again. In both cases you can select the solver type. The Direct solver is most robust (it will solve anything) but is slower than conjugate gradient solver for large problems (more than 10000 DOF or so). If the stiffness is unsymmetric you must state this in the last keyword.
- The PRINT STATE keyword causes the code to produce the contourplots.dat file. You can print DEGREES OF FREEDOM; FIELD VARIABLES (followed by a list of names); and specify how the mesh is to be printed. Note that you must write code to compute the field variables are – the list of variables is just a set of parameters that are passed to your user-subroutine. They are also used as headers in the Tecplot data file so variables can be identified while plotting contours.
- Finally there is a second PRINT command (commented out in the example provided) that activates the user-defined printing. You can define file names and any parameters that you need to control the way printing behaves – these will be passed through to the user-subroutine.

To get a feel for the various parameters and options try the following tests:

- Try increasing the magnitude of the displacement applied to the boundary nodes in the set called 'right'. You could try doing this by changing the history definition, or by supplying a numerical value instead.
- Try making the code take more than a single step (try eg 5 steps. Of course this analysis will take longer. (If you want to speed things up you can change the configuration from 'Debug' to 'Release'). You can make TECPLOT animate the sequence of deformed shapes. Of course a linear elastic code should not be used to run large deformation problems – the code will solve the equations and produce a contour plot, but the equations themselves make no sense so the solution will be meaningless.
- Try stretching the plate in the y direction instead of the x direction
- Try a different solver (try CONJUGATE GRADIENT). You won't see any difference in the solution, but the output files will contain different information.

- Try telling the solver to use a nonlinear Newton-Raphson solver instead of the linear solution. Again, you won't see any difference in the solution, but take a look at the log file to see what happened when you made the change.
- Try modifying the input file to make the plate inhomogeneous. To do this, find a line half-way down the connectivity list (eg somewhere near the 1400th element), then edit the code to include the following lines

END CONNECTIVITY
PROPERTIES
500.d0, 0.3d0
END PROPERTIES
CONNECTIVITY, hard

Then run the code again.

Notice that when you view the results in TECPLOT, you will see the interface between the two materials. Plot contours of S11 (the stress parallel to the loading axis) to see the effects of the modulus change clearly.

Note that by default each 'zone' of elements (a group that appear between CONNECTIVITY and END CONNECTIVITY) are printed in separate Tecplot 'Zones.' This is usually convenient, because it allows you to hide or activate each group of elements separately for visualization. But sometimes (eg if you are trying to make a movie) you may prefer to plot the whole mesh in a single tecplot 'zone.' You can do this by adding the following lines between the STATE PRINT and END STATE PRINT lines of the input file:

ZONES, COMBINE
Name of first zone, name of second zone, name of third zone, etc
END ZONES

The zones listed will then be combined (but note that additional nodes are generated automatically to allow field variables to be discontinuous across neighboring zones, so the node numbering scheme you will see in the output file is not the same as the one you used to define the element connectivity).

For a final set of tests, try running the Holeplate_3d_dynamic input file. This runs an explicit dynamic analysis. The contourplots.dat file will contain a large number (40) of zones – its best to animate these to see what the solution is showing. You will see stress waves propagating and dispersing through the solid.

The input file is identical to the static calculation except for the analysis section. Even this is mostly identical - the only differences are (1) no solver is used, because in explicit dynamics there is no equation solving; and (2) the parameters defining time-stepping are different. Explicit dynamic simulations need to use a very small time-step to avoid instabilities.

5. How to develop a new element

(unlike ABAQUS, in EN234FEA both material models and elements are implemented in the same user-subroutine; but you could easily write your code so that each element works with multiple material models by using the first 'element property' to select the material model).

You will follow the same procedure to implement any new element or material model in the code.

Add a new branch to your local EN234_FEA repository

(this is optional – it is not really needed for your class work but if you were doing any real software development you would not want to make changes to the master branch – you would do all your editing in a new branch and then merge it).

The new branch will contain your modifications to the code while you are working on them. This will allow you to revert to an earlier version if you need a working version of your code (or in a larger collaborative project would allow others to work on the code independently).

1. In the Eclipse Git Repositories window expand the Remote Tracking folder and right-click the repo for your fork of the EN234_FEA code (it is called origin by default but you may have renamed it when cloning your EN234_FEA fork). Select Create Branch… Give the branch a name that will help you recognize what changes it is implementing (eg Homework 1). Accept all the remaining defaults and select Finish.

Create a new file to store your element

1. In the Project Explorer window find the user_codes/src directory. Right click the new_user_element_stub.f90 file and select Copy; then paste a copy back into the same directory. Rename the copy with a name that will describe your element. In some cases you might find it more convenient to copy one of the other files – for example, the first EN234 coding homework will be to write a 2D linear elastic element. In this case it will save you time to copy and past the 3D linear elasticity element instead.

Rename the subroutines you need to write

Rename the following subroutines to something suitable for your element (don't change the arguments):

- `new_user_element_static` (…) (if you are doing an explicit dynamics computation you can delete this routine – it is used for static analysis)
- `new_user_element_dynamic` (…) (if you are doing a static computation you can delete this routine)
- `new_user_element_fieldvariables` (…) Don't forget to change the name at both the beginning and the end of each subroutine.

It's worth checking that the code compiles at this point to make sure you did everything correctly (Compile-time errors show up in the Console tab and also in the Problems tab in the window at the bottom of Eclipse)

Add calls to your new subroutines in the usrelem.f90 file

Open the file called user_element.f90 in the user_codes/src directory

1. For a static analysis, find the subroutine called `user_element_static()` and locate the lines of code below

```
else if ( element_identifier ==0) then

call new_user_element_static(lmn, element_identifier, n_nodes, node_property_list, &
    n_properties, element_properties, element_coords, length_coord_array, &
    dof_increment, dof_total, length_dof_array, &
    n_state_variables, initial_state_variables, &
    updated_state_variables,element_stiffness,element_residual, fail)
```

Copy this code exactly as it is, paste it just below the code and change the pasted code to

```
else if ( element_identifier ==identifier for your new element) then
   call name of the subroutine for your new element( leave the rest unchanged…)
```

For a dynamic analysis use the same procedure in the subroutine called `user_element_dynamic`

2. Next, find the subroutine called `user_element_fieldvariables` and locate the lines of code below

```
if ( element_identifier==0 ) then

     new_user_element_fieldvariables(lmn…
```

Copy and paste these two lines, and then edit your to make the code call your new element field variables

Again, after you are done with this step check that your code compiles. The new routines will do absolutely nothing, of course, but they will be ready to go.

Writing code for your new element

Now you are ready to insert the code for your new element. You do this by writing code to calculate the element residual force vector, and (for a static analysis) the element stiffness matrix. The comments in the code you just copied define all the input and output variables. You may find it helpful to define additional subroutines to calculate the stress and material stiffness for problems with complicated constitutive models.

Note that all the data – both the data provided, and the stiffness matrix, residual vector, and are local to the element – they just give you information about one element. EN234FEA takes care of all the book-keeping chores such as looping over all the elements, extracting the local element coordinates and degrees of freedom, as well as element state variables from storage, and inserting the element stiffness into the right place in the global stiffness matrix. You only need to code at the element level.

Here you are mostly on your own, but the el_linelast_3dbasic.f90 code will be a useful template.

Predefined subroutines

The module 'Element_Utilities' (in the file /inc/Element file usrelem.f90 contains a number of useful subroutines that will save you time while coding new elements. For details of what the subroutine arguments are see the comments in the code. They include

- Subroutines that provide Gauss integration points and weights for 1D integration 2D and 3D elements

```
initialize_integration_points(n_points, n_nodes, xi, w)
```

- Standard shape functions (and shape function derivatives) for 1D interpolation and for standard 2D and 3D elements

```
calculate_shapefunctions(xi,n_nodes,f,df)
```

- Code to invert a 2x2 and 3x3 matrix

```
invert_small(A,A_inverse,determinant)
```

There are a few other routines.

Creating an input file and testing your code

It is usually best to test your new element on a very simple problem, with one, or perhaps two elements, before trying it with a large and complicated mesh. Two examples are provided in the input files called Linear_elastic_3d.in and Linear_elastic_3d_dynamic.in. These simply apply loads to a pair of elements, for a static and dynamic analysis, respectively. You can copy and edit these files to test your own elements.

If you are coding an element for static calculations, you will need to compute both the element residual vector, and also the element stiffness matrix (for a linear analysis the residual can sometimes be left out, but it is not bad practice to code it in any case so you can combine your linear element with other nonlinear elements).

As a coding aid, EN234FEA has a capability to check that the stiffness matrix (which is the derivative of the residual vector with respect to degrees of freedom) is consistent with the residual vector. This is done by computing a numerical derivative of the residual vector (perturbing each DOF by a small amount, and the computing the change in the residual), and comparing the result with the hard-coded stiffness matrix.

To activate this feature, simply uncomment these lines in the input file

```
%      CHECK STIFFNESS, 1001
%    STOP
```

and change the element identifier 1001 to the identifier (the integer number you used to select the element in the usrelem.f90 file) corresponding to your element.

The exact and numerical stiffness matrices are printed to the 'log' file (you enter the file name when you run the code). At the end of the file the code gives a message that tells you whether the code thinks the stiffness and residual are consistent – it will also tell you if it thinks your stiffness is unsymmetric. These messages are not completely reliable – to check for errors the code needs to estimate the rounding errors in the numerical derivative, and this is difficult to do in general. It is better to check the numerical values in the tables. The numerical and exact values of the stiffness should agree to within 5-7 significant figures – if the discrepancies are bigger than this you probably have a bug.

Try running the code with the Linear_elastic_3d.in input file and check the results.

For an explicit dynamic analysis the stiffness is never computed, so this step is unnecessary.

Debugging

Eclipse has a visual debugger. To activate it simply select Debug instead of Run. You can add breakpoints, loop through the code, and so on.

I have had only partial success with the debugger on my installation of Eclipse (but I have not tried many of the options for debugging) – many variables that are defined using Fortran90 conventions do not show up properly and you cannot print their values in the Expressions window either. This can make the debugger essentially useless except for simple debugging

In desperation you may be reduced to printing values of variables to the console window or a file. The statement

```
write(6,*) 'variable name = ',variable
```

will print the value of a variable to the console window (usually the most useful option), while

```
write(IOW,*) 'variable name = ',variable
```

will print the value of a variable to the log file (More useful if you need to print huge numbers of variables. Note that the log file is buffered, and if the code crashes the buffer will not be flushed. To look for bugs that cause crashes you will need to write to the console window).

For more sophisticated Fortran output conventions see this nice discussion from Stanford.

Merge your development branch with the main branch and push changes to GitHub

1. Commit the changes to your branch (if the branch is not checked out – i.e. there is no black checkmark against the branch in the Branches/Local folder in the Git repository window – right click the branch and check it out; then use Git > Commit or else the button
2. Check out the local main branch (it should be called HEAD in the Branches/Local folder) – right click the branch and select Checkout
3. Right click the new branch you created and select Merge
4. Select Git > Push to Upstream, or use the button
5. You can delete the development branch if you don't plan to make any more changes in it – right click the branch and select 'Delete Branch.'

6. Fortran phrase-book

Fortran is one of the easiest programming languages to learn, because it can do virtually nothing other than basic low-level operations that are common to all languages. Its syntax is also very similar to MATLAB. Furthermore, you need to know only an even smaller subset of FORTRAN to be able to code elements and materials in ABAQUS or EN234FEA. Here are the things you will need:

Variable types

EN234FEA uses only four kinds of variables (you will most likely only use two of them). All variables must be explicitly declared at the top of each subroutine (see the code for examples).

To declare local variables

1. Integers:
```
integer       :: i,j,k,icount,icount2,ix,iu,is
```
2. Double precision floating point numbers (the 'prec' is the default precision; set in the Types.f90 module)
```
real (prec)   ::  xi,eta,zeta
```
3. Declaring arrays
```
real (prec)   ::  xi(3,27), w(27), x(3)
integer       ::  m(3,3)
```

Note that in FORTRAN **array indices start at 1** and increase up to the maximum value defined when declaring the variable. This differs from c and c++ but is identical to MATLAB.

Matrices are defined as A(row,column) in Fortran. For efficient code arrays should be accessed in *column major order* (i.e. vary the 'row' index faster than the 'column' index – this is the opposite of C). Optimizing compilers will usually detect and re-order loops to index arrays in the proper sequence.

It is possible to allocate arrays dynamically in Fortran90, but you should not do this inside an element or material subroutine because it will slow your code down horrendously.

4. Declaring logical variables
```
logical :: converged
```

5. Declaring character strings
```
  character ( len = 100 ) :: strin
```

The (len=xyz) specifies the length of the character string. You can create character arrays.

Fortran default variable types and arithmetic operations with mixed variable types

In EN234FEA each subroutine or function has an 'implicit none' statement at the top of the code. This means that the type of each variable must be explicitly declared in the subroutine.

In ABAQUS user subroutines you have to use the standard fortran convention for default variable types: any variable starting with the letters I through N are assumed to be integers; any others are assumed to be real (usually double precision). YOU HAVE TO BE VERY CAREFUL WITH VARIABLE NAMES.

Fortran also has a rather unusual way of doing arithmetic – the result of a calculation depends on the type of the variables in the expression. For example (with the default convention)

Result = A/B

will produce a real number for result, because A and B are both real.

Result = I/J

will be calculated as an INTEGER – it will be rounded – because I and J are both integers. If you want a real valued result you have to use

Result = dfloat(I)/dfloat(J)

Finally

Result = A/I

will be real, because A is real.

Conditional Statements

As an example

```
if (some condition) then
     code…
else if (some other condition) then
   more code
else
     more code
endif
```

Possible conditions include: >, < (greater and less than), = = (equality – note the double equals with no space), | (logical not), || (logical or), && (logical and).

Loops

Are much like MATLAB – for example

```
do n = start value of n,end value of n, (optional increment – default is 1)
        some code
enddo
```

You can nest loops…

Array and matrix operations

Fortran90 operates on vectors and matrices much like Matlab. For example

- In the statement x=sin(v) x and v can be vectors. Note that c=A*B is NOT a matrix multiplication, however – it will multiply the elements of A and B to create C.
- dot_product(u,v) and matmul(A,B) compute a general dot product and matrix multiplication and are particularly useful in FEA.
- length = size (*array*) and dims = shape(*array*) can be used to determine the length of a vector (returned as a single integer) and the dimensions of a matrix (returned as an integer vector).
- The function newarray = spread(array,dim=*d*,ncopies=*n*) will create a new array by copying an existing array. *d* specifies the new dimension; *n* is the number of copies. For example if array = (1,2,3) then A1 = spread(array,1,ncopies=3) and A2 = spread(array,1,ncopies=3) create

$$A1 = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad A2 = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}$$

- Sum(array) computes the sum of the elements in a vector (sum has some optional arguments that can make it do more complicated things too)

Subroutines

All subroutines in EN234FEA must look something like this:

```
subroutine el_linelast_3dbasic(lmn, element_identifier, n_nodes, node_property_list, &
    n_properties, element_properties, element_coords, length_coord_array, &
    dof_increment, dof_total, length_dof_array, &
    n_state_variables, initial_state_variables, &
    updated_state_variables,element_stiffness,element_residual, fail)      ! Output variables
    use Types
    use ParamIO
    !  use Globals, only: TIME,DTIME  For a time dependent problem uncomment this line to access
the time increment and total time
    use Mesh, only : node
    use Element_Utilities, only : N => shape_functions_3D
    use Element_Utilities, only : dNdxi => shape_function_derivatives_3D
    use Element_Utilities, only:  dNdx => shape_function_spatial_derivatives_3D
    use Element_Utilities, only : xi => integrationpoints_3D, w => integrationweights_3D
    use Element_Utilities, only : dxdxi => jacobian_3D
    use Element_Utilities, only : initialize_integration_points
    use Element_Utilities, only : calculate_shapefunctions
    use Element_Utilities, only : invert_small
    implicit none

    integer, intent( in )         :: lmn                    ! Element number
    integer, intent( in )         :: element_identifier     ! Flag identifying element type
    integer, intent( in )         :: n_nodes                ! # nodes on the element
    integer, intent( in )         :: n_properties           ! # properties for the element
    integer, intent( in )         :: length_coord_array     ! Total # coords
    integer, intent( in )         :: length_dof_array       ! Total # DOF
    integer, intent( in )         :: n_state_variables      ! # state variables for the element
    type (node), intent( in )     :: node_property_list(n_nodes)  ! Data structure - see below
    !  type node
    !      sequence
    !      integer :: flag                      ! Integer identifier
    !      integer :: coord_index               ! Index of first coordinate in coordinate array
    !      integer :: n_coords                  ! Total no. coordinates for the node
    !      integer :: dof_index                 ! Index of first DOF in dof array
    !      integer :: n_dof                     ! Total no. of DOF for node
    !   end type node
    !   Access these using node_property_list(k)%n_coords
    !    eg to find the number of coords for the kth node on the element
    ! Coordinates, stored as x1,(x2),(x3) for each node in turn
    real( prec ), intent( in )    :: element_coords(length_coord_array)
    ! DOF increment, stored as du1,du2,du3,du4... for each node in turn
    real( prec ), intent( in )    :: dof_increment(length_dof_array)
    ! accumulated DOF, same storage as for increment
    real( prec ), intent( in )    :: dof_total(length_dof_array)
    ! Element or material properties, stored in order listed in input file
    real( prec ), intent( in )    :: element_properties(n_properties)
    ! Element state variables at start of time increment
    real( prec ), intent( in )    :: initial_state_variables(n_state_variables
! The variables below must be defined in this subroutine
    logical, intent( out )        :: fail  ! Set to .true. to force a timestep cutback
! State variables at end of time increment
    real( prec ), intent( inout ) :: updated_state_variables(n_state_variables)
! Element stiffness (ROW,COL)
    real( prec ), intent( out ) :: element_stiffness(length_dof_array,length_dof_array)
    ! Element residual force (ROW)
    real( prec ), intent( out )   :: element_residual(length_dof_array)
```

```
    ! Local Variables
    ! Define stiffness and element residual force

   Return
 end subroutine el_linelast_3dbasic
```

The code should be self-explanatory. Some notes:

- The structure looks like a MATLAB function, except that both input and output variables are arguments (there is no [*variables*] =) as in MATLAB. You distinguish between input and output variables with the 'intent' attribute.
- Any variable declared as 'in' cannot be changed inside the subroutine.
- Any variable declared as 'out' *must* be assigned a value in the subroutine.
- Any variable declared as 'inout' can be changed or left as is.
- The 'use *xyz*' at the top of the file reference data or functions that are stored in Fortran 90 modules. They define data types; functions; or variables that are shared across different subroutines. All the modules used by EN234FEA are stored in the /modules/ sub-directory. In the element stiffness routine:
    - The 'Types' module (stored in /inc/Types.f90) declares variables like 'prec' that determines whether the code runs in single or double precision. A few standard variables like PI_D (double precision pi) are also defined, as well as 2x2 and 3x3 identity matrices.
    - The Globals module stores the time increment and total time
    - The ParamIO defines data and variables that control input/output. In particular, writing to unit number IOW will send output to the standard .out log file produced by every run.
    - The Mesh module contains all the data that specify the FEA mesh and the full solution. The 'only' statement after use Mesh means that in the element stiffness routine, only the definition of the 'node' data type is referenced. You could, if you wish, access the full FEA solution inside your element routine by adding more variables from the Mesh module.
    - The 'Element_Utilities' module defines most standard data structures and variables needed to solve a standard solid mechanics FEA problem, such as integration points, shape functions and their derivatives, and the Jacobian matrix. The syntax `N => shape_functions_3D` means that the variable N is equivalent to the variable shape_functions_3D in the Element_Utilities module. If you use the module subroutines in the 'Element Utilities' module it is important to use the module to declare all your variables too, because the subroutines use the dimensions of the arrays to determine whether to return 2D or 3D versions of the variables.
- The **`implicit none`** over-rides the default FORTRAN convention that variables starting with letters A-H and O-Z are real, and others are integers. When you code ABAQUS elements you are stuck with the standard convention, which has the advantage that you don't have to declare variable names, but on the other hand if you mis-spell a variable, the compiler will not catch it.

Writing data to files

FORTRAN has fairly rudimentary capabilities for reading and writing files. The general syntax to write a file is

Write(*unit number, format string) list of variables*
Format label Format(*description of how the output is to be formatted – equivalent to %s, %d)*

The *unit number* is an integer valued variable that is defined when the file is opened. In EN234FEA files are opened for you, and the unit numbers are provided as user subroutine arguments. You should not open and close files yourself (it will slow down the code). For debugging purposes use the variable IOW to write to the default output file (if you write to unit 6, eg Write(6,*format*) you will write to the screen on the command window).

The *format string* is a character string, eg '(I4, 1X, E12.4, A10)' that behaves much like the %s, %d formatted write in MATLAB. Here I4 means an integer length 4, 1X is a single blank space, E12.4 is a floating point number in Exponent format with total length 12 and 4 decimal places, and A10 is a character string with 10 characters.

For formatting options see this nice discussion from Stanford.

You can avoid using a format statement with
Write(*IOW,*) variable list*
This will simply use standard defaults to print integers, double precision numbers, etc. This is OK for printing a few variables but not much good for printing an array, which will be largely incomprehensible.

Some random problems I have run into with Eclipse

1. Occasionally when you double-click a .f90 file to open it, it will not highlight the code properly or let you insert breakpoints. If this happens right-click the file and say Open With… Fortran Editor.
2. Sometimes when you try to put a breakpoint into a code to debug, a big fat green break-point will show up instead of the usual small dot, and the debugger will ignore it. To fix this, right click the line number, and instead of selecting Toggle Breakpoint, select Breakpoint Types, and select C/C++ breakpoint. Then use Toggle Breakpoint in the usual way.
3. Debugger will not start and throws an error 'Launching examples.exe' has encountered a problem. Target Selection failed' Fix this by going to Run > Debug Configurations, select the Debugger tab, and select the MinGW gdb debugger in the top window (the gdb/mi option also seems to work). Select Apply, then Close.