An Object-Oriented Smartphone Application for Structural Finite Element Analysis

B.J. Mac Donald
Faculty of Engineering and Computing
Dublin City University, Dublin 9, Ireland

***Abstract*—Smartphones are becoming increasingly ubiquitous both in general society and the workplace. Recent increases in mobile processing power have shown the current generation of smartphones has equivalent processing power to a supercomputer from the early 1990s. Many industries have abandoned desktop computing and are now entirely reliant on mobile devices. Given these facts it is logical that smartphones are considered as the next platform for finite element analysis (FEA). This paper presents an architecture for a smartphone FEA application using object-oriented programming. A MVC design pattern is adopted and a demonstration FEA application for the Android smartphone platform is presented.**

Keywords—Objected-oriented programming; Finite Element Method; Java; Android

I. Introduction

Since the introduction of smartphones in 2007 they have had a profound effect on lifestyles by significantly changing the way that people live, work and learn. Smartphones have become the dominant mobile device for communication information and entertainment. In many cases smartphones (and associated tablets) have become the dominant computing platform in many industries. Smith [1] demonstrates that in excess of 46% of American adults own a smartphone and the rate of ownership rises to in excess of 60% when college graduates and high income households (in excess of $75,000) are considered. When considering these statistics, it is reasonable to assume that the majority of engineers, scientists and analysts will own, or have access to, a smartphone (or related tablet).

Many smartphone users are unaware of the computing power available in their devices and/or the potential of the smartphone as a platform for finite element analysis. Fig. 1 shows a comparison of computing power (in mega-flops) for different processors. The leftmost line (a) links the processing power of three supercomputers (Cray C1, Cray C90 and Cray Jaguar). The centre line (b) shows the processing power of desktop PC processors over time (Intel 386, Intel Pentium and Intel Core i7). The final line (c) illustrates the increase in computing power of mobile processors commonly used in Android smartphones and tablets. It is clear from fig. 1 that comparing a current high end mobile processor (e.g. Nvidia Tegra 4 which is built on ARM technology) with desktop and supercomputer processors, shows that current mobile processor capability is on par with desktop processors from circa 2008 and supercomputer processors from the early 1990's. Rajovic et al [2] discusses the development of mobile processor power in comparison to supercomputers and suggests that multicore clusters of mobile processors may actually represent the future of high powered computing.

Given that fig. 1 shows that a current mobile device is approximately equivalent in computing power to an early 1990's supercomputer or a late 2000's desktop and, considering the pioneering finite element analyses work done on these machines at the time, it is reasonable to consider current smartphones as capable of performing useful finite element analyses.

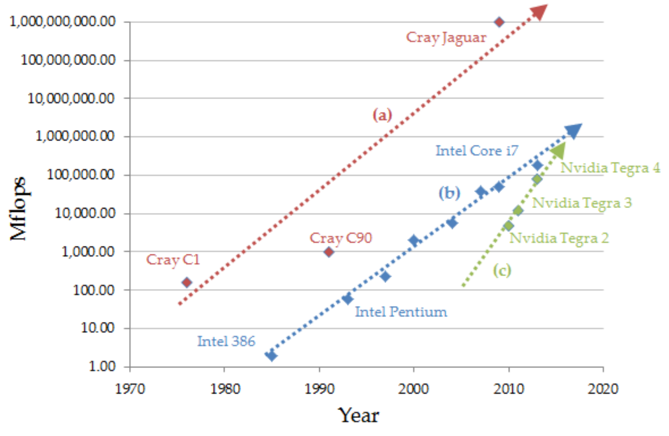


Fig. 1. Development in Computing Power (Mflops) since 1970. Trend lines show (a) supercomputers, (b) desktop PC's and (c) mobile processors.

There are currently two major operating systems available for smartphones: iOS (Apple Inc.) and Android (Google Inc.). Both of these platforms are based on objected-oriented programming languages: objective-C in the case of iOS and Java in the case of Android. Hence, any finite element code written for smartphones must be object-oriented.

Zimmermann et al. [3] described the governing principles for object-oriented finite element programming, before describing an implementation using SmallTalk [4] and C++ [5]. A number of authors [6-9] have described object-oriented implementations of the finite element method using C++.

Following the popularisation of Java in the late 1990's a number of researchers began to explore the possibilities of writing FEA codes using Java. Many researchers, however, were reluctant to engage with Java as it had a reputation for slow performance in comparison to more established non-object-oriented languages. In order to investigate this Nikishkov [10] compared the performance of a Java FEA code with a similar code written in C. It was found that with the use of proper coding and tuning it is possible to obtain similar performance from the Java and C finite element codes. In a subsequent presentation, Nikishkov [11] described the design of an object-oriented Java finite element code for the 2D and 3D analysis of elastic and elasto-plastic structural solids. The code was developed using Java 1.5 and utilised the Java3D API to allow for visualisation of the results. A user interface was not developed and model specification was handled via an input text file which was read using a scanner.

This paper describes an object-oriented smartphone application written in Android, which is effectively a subset of Java. Android was chosen as it is an open source platform which runs on many devices including smartphones, tablets, netbooks and smart televisions. Graphical user interface (GUI) design on Android is relatively straightforward as the Android API contains a multitude of classes that can easily be sub-classed to allow for complex displays and user inputs.

II. Design of the FE Application

In order to simplify the discussion that follows we will initially consider a very simple finite element application that only solves 2D truss problems. The code outlined here may easily have additional classes defined which will allow the analysis of different structural problems using different types of finite element. The requirements for the application are shown in table I.

TABLE I. Requirements for a Simple Smartphone FE Application

No	Description
1	Function without error on the majority of Android devices
2	Use the device touchscreen to allow user input
3	Allow for FEA of 2D Truss problems
4	Allow the user to define nodes by their coordinates
5	Allow the user to define linear trusses by linking two nodes
6	Allow the user to define individual element properties
7	Allow the user to place a constraint on any node in either the x or y direction
8	Allow the user to place a force on any node in either the x or y direction
9	Allow the user to easily edit the model definition by changing properties
10	Easily and efficiently solve the finite element problem and present the results
11	Allow for sharing of the results via email, social media, etc.

A Model-View-Controller (MVC) software architecture pattern was used to design the application. Fig. 2 shows an overview of the MVC pattern where we attempt to separate the representation of information from the interaction that the user has with this information. The *model* part of the pattern typically consists of data, logic and functions and, in this case, we can readily identify that our finite element classes belong here. We will designate a *model* package to contain the classes which describe the finite element model. The *view* part of the MVC pattern is used to output some representation of data to the user such as an image on a screen or a text listing etc. The *controller* part of the pattern takes input from the user and uses this input to send messages to the model or view. It is clear from fig. 2 that the user effectively interacts with the *view* part of the MVC pattern.

The view is also responsible for receiving user input and passing it to the controller. On a smartphone this is quite easy to grasp as the touchscreen on an Android device is used to both display the app and receive touch gestures. The controller receives user input from the view and acts accordingly. In most cases the controller will update the model state however it is also possible that the controller will just change the view without changing the model, for example, if a cosmetic change to the interface was requested by the user. The model stores data in its properties, implements application methods and implements the application logic. The model changes its state based on instructions from the controller. When the model changes its state it informs the view which updates accordingly.

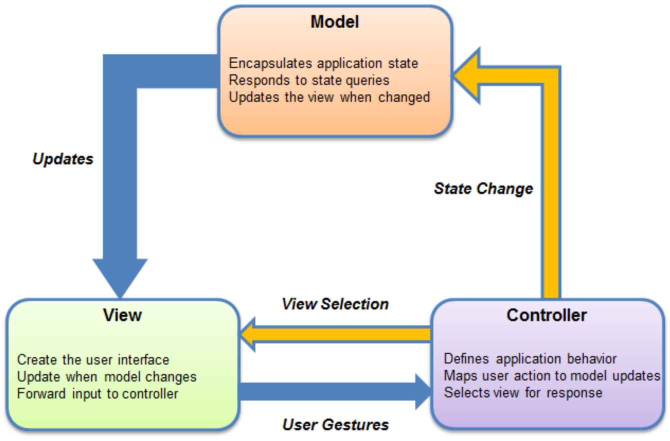

Model
Encapsulates application state
Responds to state queries
Updates the view when changed
Updates
State Change
View Selection
View
Create the user interface
Update when model changes
Forward input to controller
Controller
Defines application behavior
Maps user action to model updates
Selects view for response
User Gestures

Fig. 2. Overview of the MVC Pattern

All Android applications must have a class designated as a “Main Activity” which is the entry point into the application – much in the same way as a class with a `main()` method is for a standard Java application. In this case we have named this class `TrussActivity` and this class must extend (i.e. be a subclass of) the `android.app.Activity` class. An Activity is an application component that provides a screen with which users can interact. By sub classing `android.app.Activity` our `TrussActivity` class will gain access to all the features of the Android API and be capable of displaying information on the device screen and receiving user input via touch gestures etc.

Android and Java classes are typically organised into Packages which contain classes that have a similar function or theme as discussed above. For illustration purposes we assume the package name: `com.example.simpletruss`. The `TrussActivity` class will be placed in this package making its full name: `com.example.simpletruss.TrussActivity`. Another package is used to hold the classes that may be used to define a finite element model. These classes are Java classes and are not specific to Android and hence may be reused for any Java application. In this case, a package named “model”: `com.example.simpletruss.model` is used to hold the finite element classes. Fig. 3 shows a basic schematic of the structure of the Android FEA app: illustrating the packages used and the classes which these packages contain.

The `TrussActivity` class will take user input and create objects from the classes contained in the `com.example.simpletruss.model` package and will call methods from these classes in order to build and solve the finite element model.

The classes within the `com.example.simpletruss.model` package are largely self-explanatory. The Node class is used to create Node objects and contains helper methods associated with the manipulation of Node objects. The `Truss2D` class is a subclass of the `LineElement` class which in turn is a subclass of the `Element` class. These classes are used to create `Element` objects. The Assembly class is used to create an assembly of finite elements and contains methods to create a global stiffness matrix, global force vector and a global nodal displacement vector. The `TrussSolver` class contains methods that can take these assembled global matrices and use them to obtain a solution to a finite element problem. The `TrussPost` class contains methods that can further process the obtained solution to obtain derived results such as element stress and strain. The `FeConstants` class contains a list of symbolic constants that may be used by all other classes within the package.

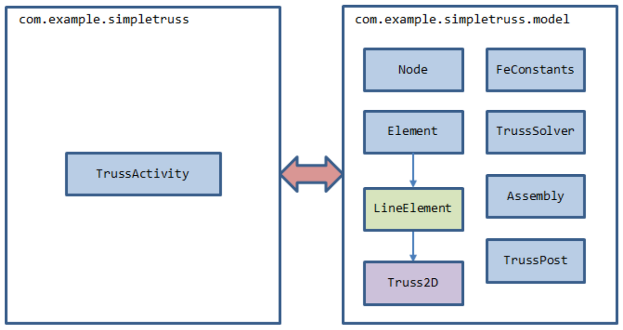


Fig. 3. Schematic of the Packages and Classes in the Android Application

By resolving the MVC pattern shown in fig. 1 with the schematic shown in fig. 2 it is clear that the com.example.simpletruss.model package will function exactly as the model is described in the MVC pattern. The TrussActivity class provides a method of linking into the Android API by sub classing android.app.Activity. Each Activity must implement the onCreate() method inherited from the superclass. In its simplest form the onCreate() method will be:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_layout);
}
```

The final line in the above code snippet calls the superclass method `setContentView()` to set the View that will be shown to the user when the application is started. This layout is usually specified in an XML layout file named main_layout.XML. This XML layout file may be edited to display the buttons, text fields, checkboxes, images, etc. that make up the applications GUI.

Effectively, the onCreate() method links us into the Android API via android.app.Activity and provides us with a View using setContentView() via android.view.View.

The controller part of the MVC pattern will consist of the other methods contained in TrussActivity which are not inherited from the superclass. These are methods which are custom written for the FE application. These methods are summarised in table II.

TABLE II. CONTROLLER METHODS IN TRUSSACTIVITY

Method	Description
addNode()	Creates a Node object using user input from a dialog box
deleteNode()	Deletes a Node object from the database using a dialog box
addElement()	Creates an Element object using a dialog box
deleteElement()	Deletes an Element object from the database
addConstraint()	Sets a constraint on a Node object using a dialog box
deleteConstraint()	Modifies a constraint on a Node object
addForce()	Sets a force on a Node object using a dialog box for user input
deleteForce()	Modifies a force on a Node object
calculate()	Uses the database of Node and Element objects to create an assembly of finite elements, solves the global problem and then creates a new View to display the results, simultaneously saves the results to a text file for sharing

Each of the methods described in Table II performs two basic functions: instructing the view what view to provide (add a node dialog, delete a force dialog, results screen etc.) and processing user input from this view and using it to change the state of the model (add a new node object, change the force on a node object, etc.).

So, in summary, the com.example.simpletruss.model package contains the *Model*, the onCreate() method in TrussActivity class and its associated XML files contain the *View* and the other methods in TrussActivity class define the *controller*. This is illustrated in fig. 4

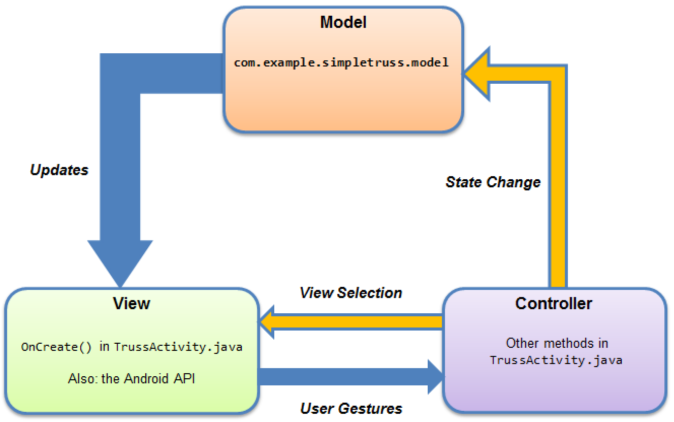

Model
com.example.simpletruss.model
Updates
State Change
View Selection
View
OnCreate() in TrussActivity.java
Also: the Android API
Controller
Other methods in TrussActivity.java
User Gestures

Fig. 4. A MVC Implementation for the FE Smartphone Application

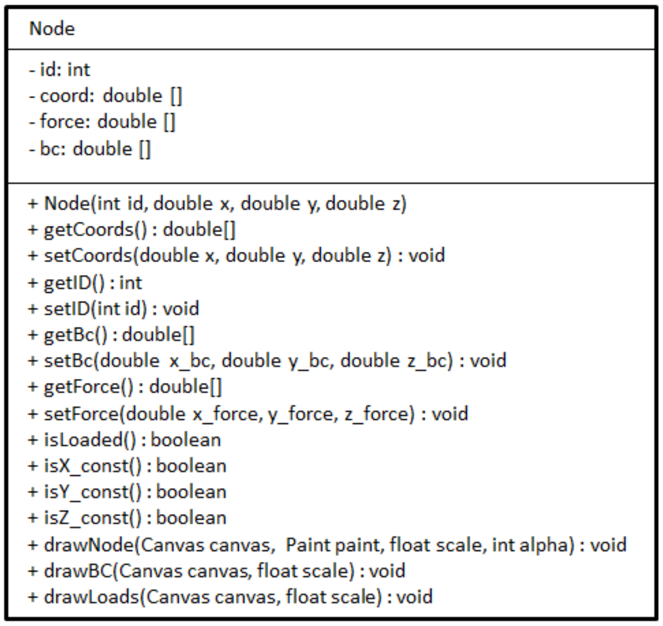
III. Model

The subsections below describe the classes in the model package which is responsible for building the finite element model and solving the global problem.

A. *Node*

The Node class is common to all finite element types and will be unchanged regardless of the element type used. A description of the Node class is shown in table III.

TABLE III. Description of the Node Class



Node
- id: int - coord: double [] - force: double [] - bc: double []
+ Node(int id, double x, double y, double z) + getCoords() : double[] + setCoords(double x, double y, double z) : void + getID() : int + setID(int id) : void + getBc() : double[] + setBc(double x_bc, double y_bc, double z_bc) : void + getForce() : double[] + setForce(double x_force, y_force, z_force) : void + isLoaded() : boolean + isX_const() : boolean + isY_const() : boolean + isZ_const() : boolean + drawNode(Canvas canvas, Paint paint, float scale, int alpha) : void + drawBC(Canvas canvas, float scale) : void + drawLoads(Canvas canvas, float scale) : void

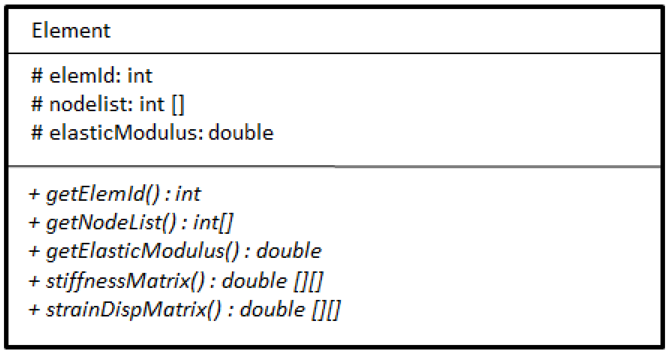
Each node object has an id, array of global coordinates, array of applied forces and an array of boundary conditions. Each of these arrays has three members of double precision numbers: for the x, y and z directions.

The Node constructor creates a node object using its id and its x, y and z coordinates. Nodes may be defined in 2D space by setting z equal to zero. Public getter and setter methods are provided in order to allow for reporting and modification of a nodes properties. A number of helper methods are provided to quickly determine if a node has an applied load or boundary condition. These methods return a Boolean value and are generally used to aid in the graphical display of loads and boundary conditions. Finally, helper methods are provided which allow for display of the node and its associated applied forces and boundary conditions in the applications GUI.

B. *Element Classes*

The Element class is an abstract class for all finite element types. A description of the Element class is shown in table IV. Each element object must have an id, a list of nodes that define the element and an elastic modulus. Several abstract methods (shown in italics) are provided which must be implemented by any subclasses: these methods provide for reporting of element properties and assembly of the elements stiffness matrix and strain displacement matrix.

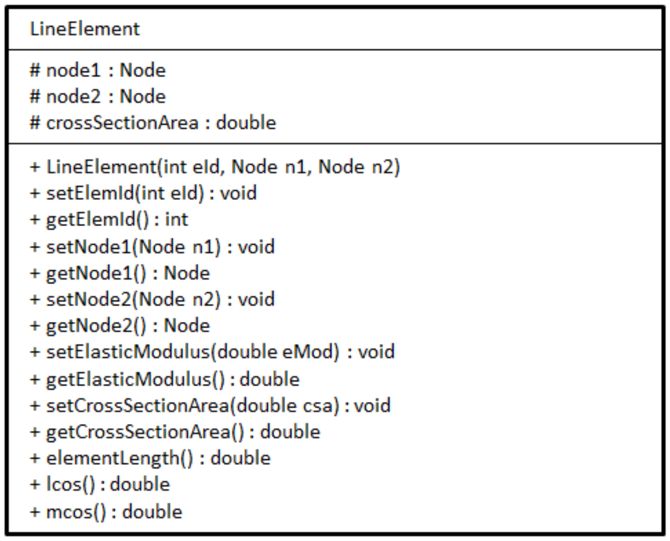
TABLE IV. Description of the Element Abstract Class



Element
elemId: int # nodelist: int [] # elasticModulus: double
+ getElemId() : int *+ getNodeList() : int[]* *+ getElasticModulus() : double* *+ stiffnessMatrix() : double [][]* *+ strainDispMatrix() : double [][]*

The LineElement class is a subclass of the Element class and, as such must implement its abstract methods. Table V shows a description of the LineElement class.

TABLE V. Description of the LineElement Class



LineElement
node1 : Node # node2 : Node # crossSectionArea : double
+ LineElement(int eId, Node n1, Node n2) + setElemId(int eId) : void + getElemId() : int + setNode1(Node n1) : void + getNode1() : Node + setNode2(Node n2) : void + getNode2() : Node + setElasticModulus(double eMod) : void + getElasticModulus() : double + setCrossSectionArea(double csa) : void + getCrossSectionArea() : double + elementLength() : double + lcos() : double + mcos() : double

Each LineElement object is defined by two Node objects and its cross sectional area. The constructor creates LineElement objects using this data. Several getter and setter methods are provided to allow for reporting and modification of element properties. Finally, three helper methods are provided which calculate the element length and its direction cosines, *l* and *m*.

The Truss2D class is a subclass of both LineElement and Element (via the class hierarchy). A 2D truss is obviously a line element and so inherits all the properties and methods of its superclass. The Truss2D class is primarily concerned with implementing methods specific to 2D truss finite elements. Table V shows a description of the Truss2D class. Since most of the methods required for a 2D truss have already been implemented in the superclass's, the Truss2D class is relatively short. It essentially consists of a constructor which simply calls the constructor of the superclass and two methods which calculate the element stiffness matrix and strain displacement matrix.

TABLE VI. DESCRIPTION OF THE TRUSS2D CLASS

Truss2D
- stiffnessMatrix : double [4][4] - strainDispMatrix : double [1][4]
+ Truss2D(int eId, Node n1, Node n2) + stiffnessMatrix() : double [4][4] + strainDispMatrix() : double [1][4]

C. Assembly Class

The Assembly class essentially consists of five methods which assemble the global problem equations. The global stiffness matrix is assembled from the individual element stiffness matrices and placed into a 2D array of double precision numbers. Similarly the global force vector and global displacement vector are assembled by interrogating each element to find its constituent Node objects and their relevant force and boundary condition data. Two further methods are used to assemble global data which will be useful during post-processing of results. An ArrayList of element strain-displacement matrices and an ArrayList of element elastic-modulii are produced by calling these methods. The Assembly class is written in a non-element specific manner so that it may be used with any element type, not just the truss elements being considered here.

D. TrussSolverClass

The TrussSolver class contains one method named calculateDisplacements() which returns the solved nodal displacement vector to the calling method or class. A direct equation solver performs solution of the system equation using symmetric LDU decomposition of the matrix.

E. TrussPost Class

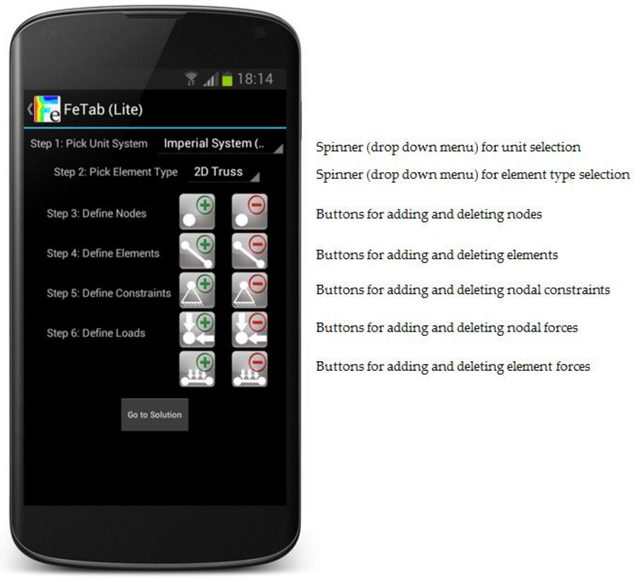
The TrussPost class contains a number of methods for post-processing the results from a truss analysis. The strains() method is used to return an array of doubles which effectively gives the strain in each element in the finite element model. Similarly, method stress() provides an array listing the axial stress in each element in the finite element model. Finally, method reactionForces() is used to return an array listing the reaction forces at each node in the finite element model.

F. A Note on the Model Classes

Clearly, there are several possibilities available for class construction and interaction when using an objected-oriented approach. The above description attempts to take the four principles of object-oriented design (Encapsulation, inheritance, polymorphism and abstraction) into account at all times. It could be argued that the Assembly, Solver and Post-Processor classes could be either combined into one class, or, are not really classes at all and their methods should be combined into other classes (e.g. one of the element classes). Alternatively, these methods could be placed in a class which contains only a list of static methods and thus does not require instantiation in order to call the methods. Both of these strategies, however, would remove the flexibility of the software and make it more difficult to add additional element types to the finite element application.

IV. VIEW

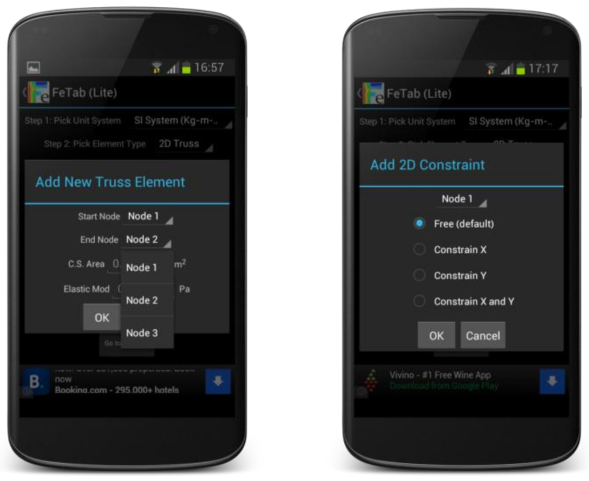
As described in section II, each screen in Android is represented using an XML layout file. It is also possible to create the layout dynamically during program execution but, in most cases, it is preferable to define an XML layout in advance. Fig. 5 shows the main screen used for building the finite element model in the completed smartphone application.


18:14
FeTab (Lite)
Step 3: Define Nodes
Step 4: Define Elements
Step 5: Define Constraints
Step 6: Define Loads
Go to Solution
Spinner (drop down menu) for unit selection
Spinner (drop down menu) for element type selection
Buttons for adding and deleting nodes
Buttons for adding and deleting elements
Buttons for adding and deleting nodal constraints
Buttons for adding and deleting nodal forces
Buttons for adding and deleting element forces

Step 1: Pick Unit System Imperial System (..
Step 2: Pick Element Type 2D Truss

Fig. 5. Graphical User Interface (GUI) for the Smartphone Application Pre-Processor

The screen layout is divided into a number of steps that the user is required to complete in order to successfully build a finite element model. The layout was constructed in this manner in order to avoid user confusion and also, as one of the aims of the application was for it to be used as an educational tool to teach FEA to new users. In the first step a drop down menu (known as a “Spinner” in Android) is used to capture the user’s preference in terms of unit system. The selected unit system is used to prompt the user for input quantities during the model generation and also during the display of results. The user is offered three choices: no units (which is the default), the SI system (Kg-m-sec) or the Imperial system (lb-ft-sec). Step 2 requires the user to pick an element type: currently there are three options available: 1DTruss, 2DTruss and Beam. The class system for a 2D truss analysis was discussed in section III. A 1D truss can be easily formed by simply setting the appropriate coordinates and DOF to zero. A beam element was implemented by adding additional classes to the structure discussed in section III and, for the sake of clarity, will not be discussed here. Step 3 requires the user to specify nodal coordinates. Touching on either the add node or delete node button opens a dialog box which allows the user to define the nodal coordinates. Similarly the add element, delete elements, add constraint, delete constraint, add nodal force and delete nodal force buttons all open appropriate dialog boxes for the user to interact with. The two lower buttons allow for the application of distributed loads if a beam element type has been selected – if a truss element is selected then these buttons will display a warning. Fig. 6 shows examples of the "Add Node" and "Add Element" dialogs.



Step 1: Pick Unit System SI System (Kg-m-...
Step 2: Pick Element Type 2D Truss

Add New Truss Element
Start Node Node 1
End Node Node 2
C.S. Area m²
Elastic Mod Pa
Node 1
Node 2
Node 3

Add 2D Constraint
Node 1
Free (default)
Constrain X
Constrain Y
Constrain X and Y

Fig. 6. Dialog Boxes are Used to Capture User Input

During the XML definition of the buttons shown in fig. 5 a method name in TrussActivity is required in order to link the button to that method. For example the "Add a Node" button definition contains a reference to the addNode() method in TrussActivity. When the button is touched/clicked then the relevant method is called and the object reference of the View calling the method is passed as a parameter to the method.

The full suite of Android's user interface was utilised to capture input from the user: including spinners, checkboxes, radio-buttons, textboxes etc. Touching the application icon at the top of the screen slides a menu out from the left hand side which allows the user to navigate through the application, as shown in fig. 7.

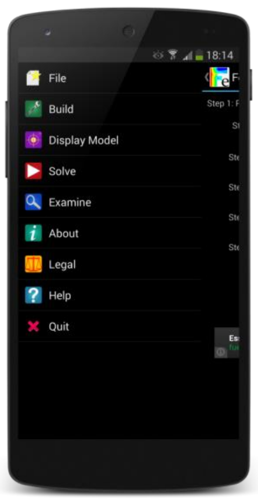


Fig. 7. Smartphone FE Application Navigation Menu

Touching the "File" button allows the user to load or save a model and clear the database. The "Build" button brings up the pre-processor screen shown in fig. 5. The "Display Model" button is used to show a graphical representation of the model, as shown in fig. 8. The "Examine" button is used for post-processing the results from the finite element model. The other buttons in fig. 7 are largely self-explanatory.

Each of the screens described above are created using XML layout files which specify the relative position of the various UI elements. These layouts are displayed by the corresponding Android activity class when required. In some cases, such as with dialogs, the display is created dynamically using only Java code without the need for a XML layout to be defined in advance. This is achieved using one of the many "builder" classes provided with the Android API. The graphical display of the model is also created dynamically by filling an empty frame layout with a Canvas object when the user requests the model be displayed.

Fig. 8 shows a typical graphical display from a 2D Truss problem. In this case three nodes and two elements have been used. Node numbers are displayed near the associated nodes. Constraint and load symbols are placed on relevant nodes, using the helper methods described in section III. A facility for zooming in/out and an option to fit the finite element model to the screen are provided in the lower right corner of the GUI.

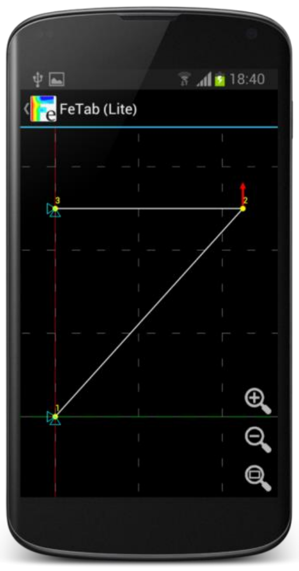


Fig. 8. Graphical Display of a 2D Truss Problem. Note display of constraint symbols on left hand side and load arrow symbols on the right.

V. CONTROLLER

The TrussActivity class is the main activity for a truss analysis. As mentioned above, the first task of TrussActivity is to call the onCreate() method from its superclass. This method is called when the application is started and is responsible for providing the View for the Activity by linking to the appropriate XML layout file.

The main task of TrussActivity is to act as controller in the MVC pattern and to take user input in order to use the Model classes to construct a finite element model. Table VII shows a description of the TrussActivity class, focusing on the methods dealing with control. A number of EditText object references are initially described as private class variables. EditText's are editable text boxes that are used to obtain user input. In this case they are required to capture nodal coordinates, element properties, etc. Two ArrayList objects are defined which effectively act as the finite element model database. The nodes ArrayList holds a list of currently defined Node objects and the elements ArrayList holds a list of currently defined LineElement objects. ArrayLists are effectively mutable arrays and so allow for the addition and subtraction of objects from the list as required. Two integer variables are defined in order to conveniently hold the number of currently defined nodes and elements.

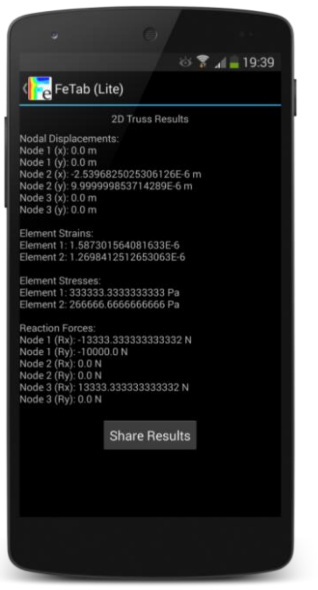
TABLE VII. DESCRIPTION OF THE TRUSSACTIVITY CLASS

TrussActivity
- xNode : EditText - yNode : EditText - startNode : EditText - endNode : EditText - area : EditText - elasticModulus : EditText - deleteNode : EditText - deleteElement : EditText - xConst : double - yConst : double - nodes : ArrayList<Node> - elements : ArrayList<LineElement> - numNodes : int - numElements : int
onCreate(Bundle savedInstance) : void + addNode(View v) : void + delNode(View v) : void + addElement(View v) : void + deleteElement(View v) : void + addConstraint(View v) : void + deleteConstraint(View v) : void + addForce(View v) : void + deleteForce(View v) : void + calculate (View v) : void + printTrussResults(View v) : void

The addNode() method is triggered by the user touching the "Add a Node" button on the main screen (fig. 5). An object reference to the View that requested the method to be called is passed in as the parameter v. This reference is required as it tells the addNode() method how/where to update the View if required.

Each of the methods shown below the addNode() method in table VII follow a standard procedure so the addNode() method will be used to illustrate this procedure. The addNode() method begins by creating a dialog box in the current View in order to obtain user input. The method then sets up a listener to listen for either the cancel or OK buttons in the dialog box to be touched by the user. If the cancel button is touched then the dialog is simply dismissed and control is returned to the calling method. If the OK button is touched then data entered by the user is checked for viability. If the data is not viable then a message is displayed to the user explaining why this is the case. If the data is viable then a new Node object is created using the object constructor in the Node class. This Node object is then added to the nodes ArrayList and the numNodes variable is incremented by 1 before returning control to the calling method.

Some of the other methods require more checks before displaying a dialog requesting user input. The addElement() method, for example, first checks that at least two Node objects have been defined before allowing the user to proceed. In each case where a problem is encountered an explanatory message is presented to the user.



```
2D Truss Results

Nodal Displacements:
Node 1 (x): 0.0 m
Node 1 (y): 0.0 m
Node 2 (x): -2.5396825025306126E-6 m
Node 2 (y): 9.999999853714289E-6 m
Node 3 (x): 0.0 m
Node 3 (y): 0.0 m

Element Strains:
Element 1: 1.587301564081633E-6
Element 2: 1.2698412512653063E-6

Element Stresses:
Element 1: 333333.3333333333 Pa
Element 2: 266666.6666666666 Pa

Reaction Forces:
Node 1 (Rx): -13333.333333333332 N
Node 1 (Ry): -10000.0 N
Node 2 (Rx): 0.0 N
Node 2 (Ry): 0.0 N
Node 3 (Rx): 13333.333333333332 N
Node 3 (Ry): 0.0 N
```

Fig. 9. Typical Results Display

The calculate() method begins solution of the finite element model. Before attempting to form the global assembly a number of checks are carried out to ensure the model is ready for solution: at least one element is defined, at least on DOF is constrained, at least one nodal force has been specified, etc. In each case an appropriate message is displayed to the user if a problem is encountered. If no problems are found then assembly of the global system of equations proceeds as described in section III. The assembled problem is then solved using the TrussSolver class which returns an Array containing the solved global displacement Vector. A quick check is performed to ensure that the returned array is not empty (indicating a failed solution). If this is the case then a message regarding the mathematical un-stability of the finite element model, together with some advice on how to fix the model is presented to the user. If the global displacement vector is valid then the printTrussResults() method is called and results are automatically post-processed and displayed to the user. Fig. 9 shows a typical display of results from a simple 2D truss analysis.

VI. Conclusion

The architecture of a demonstration finite element analysis application for an Android smartphone has been presented. The application has been designed according to object-oriented principles using a MVC design pattern. Smartphone user interfaces provide exciting opportunities to revolutionise the generation and analysis of finite element models. In this case the objective was to produce a functioning finite element application which could also be used as an educational tool to teach new users basic FEA principles. The Android platform makes it relatively easy to design an intuitive and educational user interface. The architecture provided here can easily be expanded to include more complex elements and analysis capabilities. The demonstration application is available for free download [12].

References

[1] A. Smith, 46% of American Adults are now Smartphone Owners, Pew Internet, 2012 (http://pewinternet.org/Reports/2012/Smartphone-Update-2012.aspx)

[2] N. Rajovic, P. Carpenter, I. Gelado, N. Puzovic and A. Ramirez, Are Mobile Processors Ready for HPC?, edaWorkshop13, Dresden, Germany, May 14-16, 2013.

[3] T. Zimmermann, Y. Dubois-Pélerin and P. Bomme, Object-oriented Finite Element Programming: I. Governing Principles, Computer Methods in Applied Mechanics and Engineering, 1992, 98, No. 2, pp. 291-303.

[4] T. Zimmermann, Y. Dubois-Pélerin and P. Bomme, Object-oriented Finite Element Programming: II. A Prototype Program in Smalltalk, Computer Methods in Applied Mechanics and Engineering, 1992, 98, No.3, pp. 361-397.

[5] T. Zimmermann, Y. Dubois-Pélerin and P. Bomme, Object-oriented Finite Element Programming: II. An Efficient Implementation in C++, Computer Methods in Applied Mechanics and Engineering, 1993, 108, No.1-2, pp. 165-183.

[6] P. Donescu & Tod. A Laursen, A Generalized Object-Oriented Approach to Solving Ordinary and Partial Differential Equations Using Finite Elements, Finite Elements in Analysis and Design, 1996, 22, pp. 93-107

[7] J. Besson & R. Foerch, Large Scale Object-oriented Finite Element Code Design, Computer Methods in Applied Mechanics and Engineering, 1997, 142, pp. 165-187.

[8] G.C. Archer, G. Fenves & C. Thewalt, A New Object-oriented Finite Element Analysis Program Architecture, Computers and Structures, 1999, 70, pp. 63-75

[9] B. Patzák & Z. Bittnar, Design of Object-oriented Finite Element Code, Advances in Engineering Software, 2001, 32, 759-767

[10] G.P.Nikishkov, Yu.G.Nikishkov and V.V.Savchenko, Comparison of C And Java Performance In Finite Element Computations, Computers and Structures, 2003, 81, pp. 2401-2408

[11] G.P.Nikishkov, Object oriented design of a finite element code in Java. Computer Modeling in Engineering and Sciences, 2006, 11, No. 2, pp. 81-90

[12] https://play.google.com/store/apps/details?id=ie.jion.fetab