The OpenSees Command Language Primer

Version 1.1 - Preliminary Draft
December 15, 2000
Frank McKenna and Gregory L. Fenves
PEER, University of California at Berkeley

1 Introduction

This document is intended to outline the rudimentary commands currently available with opensees. opensees is an interpreter being developed for use with OpenSees. OpenSees is an object-oriented framework under construction for finite element analysis. OpenSees's intended users are in the research community. A key feature of OpenSees is the interchangeability of components and the ability to integrate existing libraries and new components into the framework (not just new element classes) without the need to change the existing code. Core components, that is the abstract base classes, define the minimal interface (minimal to make adding new component classes easier but large enough to ensure all that is required can be accommodated).

OpenSees is comprised of a set of modules to perform creation of the finite element model, specification of an analysis procedure, selection of quantities to be monitored during the analysis, and the output of results. In each finite element analysis, an analyst constructs 4 main types of objects, as shown in figure 1:

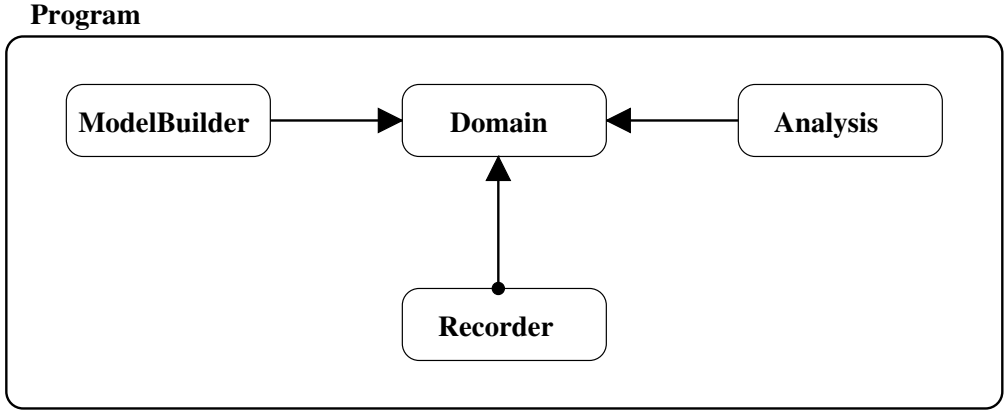

Figure 1: Main Objects in an Analysis

1. **ModelBuilder**: As in any finite element analysis, the analyst's first step is to subdivide the body under study into elements and nodes, to define loads acting on the elements and nodes, and to define constraints acting on the nodes. The ModelBuilder

is the object in the program responsible for building the Element, Node, LoadPattern, TimeSeries, Load and Constraint objects.

2. **Domain**: The Domain object is responsible for storing the objects created by the ModelBuilder object and for providing the Analysis and Recorder objects access to these objects.

3. **Analysis**: Once the analyst has defined the model, the next step is to define the analysis that is to be performed on the model. This may vary from a simple static linear analysis to a transient non-linear analysis. The Analysis object is responsible for performing the analysis. In OpenSees each Analysis object is composed of several component objects, which define how the analysis is performed. The component classes consist of the following: SolutionAlgorithm, Integrator, ConstraintHandler, DOF_Numberer, SystemOfEqn, Solver, and AnalysisModel.

4. **Recorder**: Once the model and analysis objects have been defined, the analyst has the option of specifying what is to be monitored during the analysis. This, for example, could be the displacement history at a node in a transient analysis or the entire state of the model at each step in the solution procedure. Several Recorder objects are created by the analyst to monitor the analysis.

The main abstractions of OpenSees will be explained using opensees. opensees is an interpreter for openseesTcl, an extension of the Tcl scripting language. Tcl is a string based procedural command language which allows substitution, loops, mathematical expressions, and procedures. opensees adds commands to Tcl for finite element analysis. Each of these commands is associated (bound) with a C++ procedure that is provided. It is this procedure that is called upon by the interpreter to parse the command. In this document we outline only those commands which have been added to Tcl by opensees. Following the outline of the new commands, an example openseesTcl script is presented which performs an analysis of a simple two dimensional truss model.

2 Tcl Basics

The basic syntax for a Tcl command is

```
command arg1 arg2 aropensees ...
```

where command is the name of the Tcl command or is a user defined procedure and arg1 arg2 ... are the arguments for the command. Tcl allows any argument to be a nested command:

```
command [nested command 1] [nested command 2] ...
```

where the [ ] are used to delimit the nested commands. The Tcl interpreter will first evaluate the nested commands and will evaluate the outer command with the result of the nested commands.

The most basic command in Tcl is the set command:

```
set variable value
```

which takes two arguments, the variables name and the value it is to be assigned. Value may be a string or number (in Tcl everything is treated as a string). To obtain the value of a variable the $ operator is used.

To evaluate mathematical expressions the expr command is used:

```
expr expression
```

where the expression may be any valid mathematical expression used in the C programming language. Tcl allows variable substitution in the expression.

Double quotes and braces can be used to group strings into one argument for a command. The difference is that quotes allow substitution to occur in the group, where as braces do not, for example:

```
set a 5
-> 5
puts "a is $a"
-> a is 5
puts {a is $a}
-> a is $a
```

Procedures are defined using the command proc:

```
proc name args body
```

where name is the name of the Tcl procedure created, args is the procedure arguments and body is the body of the procedure, for example:

```
set a 5
proc sum {arg1 arg2} {
return [expr $arg1 + $arg2]
}
sum $a $a
->10
```

Tcl also allows for loops and conditional evaluation. For more details see 'Practical Programming in Tcl and Tk' by Brent B. Welch.

3 Notation

For the rest of this document the following notation will be used. Input values are a string unless terminated by a ?, in which case an integer or floating point number is to be provided. Optional values are identified in enclosing < > braces. When specifying a quantity of x values are required, the command line contains (x values?). An arbitrary number of input values is indicated with the dotdotdot notation, i.e. value1? value2? ... .

4 The model Command

```
model modelBuilderType <specific model builder args>
```

The model command has at least one argument which identifies the type of ModelBuilder object to be constructed. Currently there is only one type of ModelBuilder accepted, that of type BasicBuilder.

```
model BasicBuilder -ndm ndm? <-ndf ndf?>
```

The command for constructing a BasicBuilder object contains additional arguments. The string -ndm followed by an integer defining the dimension of the problem, i.e. 1, 2 or 3-d. By default the number of degrees-of-freedom at a node (ndf) depend on the value of ndm (ndm=1, ndf=1; ndm=2, ndf=3; ndm=3, ndf=6). An optional string -ndf followed by an integer defining the number of degrees associated with each node can also be specified if the analyst should require degrees of freedom different from the defaults.

The construction of the BasicBuilder object adds additional commands to the opensees language. These additional commands allow for the construction of Nodes, Elements, Load-Patterns, TimeSeries, Loads and Constraints. The additional commands are as follows:

4.1 The node Command

```
node nodeTag? (ndm coordinates?) <-mass (ndf values?)>
```

The node command is used to construct a Node object. The first argument to the node command defines the integer tag that uniquely identifies the node object among all other nodes in the model. Following the tag argument, ndm nodal coordinates must be provided to define the spatial location of the Node. An optional string -mass accompanied by ndf mass terms following the specification of the coordinates allows the analyst the option of associating nodal mass with the Node.

4.2 The mass Command

```
mass nodeTag? (ndf values?)
```

The mass command is used to set the mass at a node. The first argument to the node command defines the integer tag that uniquely identifies the node for which the mass will be set. Following the tag argument, ndf mass terms are specified, where ndf is the number of degrees of freedom per node in the model.

4.3 The uniaxialMaterial Command

```
uniaxialMaterial materialType <specific material args>
```

The uniaxialMaterial command is used to construct a UniaxialMaterial object. UniaxialMaterial objects represent uniaxial stress-strain (or force-deformation) relationships. The command has at least one argument, the string materialType, which identifies the type of material being constructed. Currently the following types are permitted: Elastic, ElasticPP, ElasticPPGap, Parallel, Series, Hardening, Steel01, Concrete01, Hysteretic, and Viscous. The commands for specifying each type are as outlined below.

The valid queries to any uniaxial material when creating an ElementRecorder are 'strain', 'stress', and 'tangent'.

4.3.1 Elastic Material

```
uniaxialMaterial Elastic matTag? E? <eta?>
```

To construct an elastic uniaxial material with a tangent of E and optional damping tangent of eta. The argument matTag is used to uniquely identify this UniaxialMaterial object among UniaxialMaterial objects in the BasicBuilder object.

4.3.2 Elastic-Perfectly Plastic Material

```
uniaxialMaterial ElasticPP matTag? E? ep?
```

To construct an elastic perfectly plastic uniaxial material with an elastic tangent of E which reaches the plastic state at a strain of ep. The argument matTag is used to uniquely identify this UniaxialMaterial object among UniaxialMaterial objects in the BasicBuilder object.

4.3.3 Elastic-Perfectly Plastic Gap Material

```
uniaxialMaterial ElasticPPGap matTag? E? fy? gap?
```

To construct an elastic perfectly plastic gap uniaxial material with an elastic tangent of E, which reaches the plastic state at a stress of fy. The initial gap of the model is given by the argument gap. The argument matTag is used to uniquely identify this UniaxialMaterial object among UniaxialMaterial objects in the BasicBuilder object.

4.3.4 Parallel Material

```
uniaxialMaterial Parallel matTag? tag1? tag2? ... <-min min?> <-max max?>
```

To construct a parallel material model made up of an arbitrary number of previously constructed UniaxialMaterial objects, which are identified by the tags tag1 tag2 ... . In a parallel model, strains are equal and stresses and tangents are additive. Specification of minimum and maximum failure strains through the -min and -max switches is optional. The argument matTag is used to uniquely identify this UniaxialMaterial object among UniaxialMaterial objects in the BasicBuilder object.

4.3.5 Series Material

```
uniaxialMaterial Series matTag? tag1? tag2? ...
```

To construct a series material model made up of an arbitrary number of previously constructed UniaxialMaterial objects, which are identified by the tags tag1 tag2 ... . In a series model, stresses are equal and strains and flexibilities are additive. The argument matTag is used to uniquely identify this UniaxialMaterial object among UniaxialMaterial objects in the BasicBuilder object.

4.3.6 Hardening Material

```
uniaxialMaterial Hardening matTag? E? sigmaY? H_iso? H_kin?
```

To construct a uniaxial material model with combined linear kinematic and isotropic hardening. The model contains a yield stress of sigmaY, an elastic modulus of E, an isotropic hardening modulus of H_iso, and a kinematic hardening modulus of H_kin. The argument matTag is used to uniquely identify this UniaxialMaterial object among UniaxialMaterial objects in the BasicBuilder object.

4.3.7 Steel01 Material

```
uniaxialMaterial Steel01 matTag? fy? E0? b? <a1? a2? a3? a4?>
                        <-min min?> <-max max?>
```

To construct a uniaxial bilinear steel model with kinematic hardening and optional isotropic hardening described by a non-linear evolution equation. The model contains a yield strength of fy, an initial elastic tangent of E0, and a hardening ratio of b. The optional parameters a1, a2, a3, and a4 control the amount of isotropic hardening (default values are provided for no isitropic hardening). Specification of minimum and maximum failure strains through the -min and -max switches is optional and must appear after the specification of the isoptropic hardening parameters, if present. The argument matTag is used to uniquely identify this UniaxialMaterial object among UniaxialMaterial objects in the BasicBuilder object.

4.3.8 Concrete01 Material

```
uniaxialMaterial Concrete01 matTag? fpc? epsc0? fpcu? epscu? <-min min?> <-max max?>
```

To construct a uniaxial Kent-Scott-Park concrete model with degraded linear unloading/reloading stiffness according to the work of Karsan-Jirsa and no strength in tension. The model contains a compressive strength of fpc, a strain at the compressive strength of epsc0, a crushing strength of fpcu, and a strain at the crushing strength of epscu. Compressive concrete parameters should be input as negative values for this model. Specification of minimum and maximum failure strains through the -min and -max switches is optional. The argument matTag is used to uniquely identify this UniaxialMaterial object among UniaxialMaterial objects in the BasicBuilder object.

4.3.9 Hysteretic Material

```
uniaxialMaterial Hysteretic matTag? s1p? e1p? s2p? e2p? <s3p? e3p?>
    s1n? e1n? s2n? e2n? <s3n? e3n?> pinchX? pinchY? damage1? damage2? <beta?>
```

To construct a bilinear hysteretic model with pinching of force and deformation, damage due to ductility and energy, and degraded unloading stiffness based on ductility. Points on the backbone are specifed by the arguments s1p, e1p, etc., where s indicates force, e indicates deformation, 1 is the first point on the backbone (yield), 2 the second point, and 3 indicates an optional third point for a trilinear backbone. p indicates positive backbone points, and n negative backbone points. Note that negative backbone points should be entered as negative numeric values. The pinching factors pinchX and pinchY indicate the amount of pinching of deformation and force, respectively, during reloading. The factors damage1 and damage2 are for damage due to ductility, $D_1(\mu - 1)$, and energy, $D_2(\frac{E_i}{E_{ult}})$, respectively. The optional parameter beta is a power used to determine degraded unloading stiffness based on ductility, $\mu^{-\beta}$.

4.4 The nDMaterial Command

```
nDMaterial materialType <specific material args>
```

The nDMaterial command is used to construct an NDMaterial object. NDMaterial objects represent stress-strain relationships at the integration points of continuum and force-deformation elements. The command has at least one argument, the material type. Currently the following types are permitted: ElasticIsotropic and J2Plastciity continuum models and one force-deformation model Bidirectional. The commands for specifying each type are as outlined below.

The valid queries to any ND material when creating an ElementRecorder are 'strain', 'stress', and 'tangent'.

4.4.1 Elastic Isotropic Material

```
nDMaterial ElasticIsotropic matTag? E? v?
```

To construct an ElasticIsotropic material object with elastic modulus E and Poisson ratio v. The argument matTag is used to uniquely identify this NDMaterial object among NDMaterial objects in the BasicBuilder object.

The material formulations for the ElasticIsotropic object are 'PlaneStrain2D' and 'PlaneStress2D'. These are the valid strings that can be passed to the quad element for the type parameter. Other types will be added as more continuum elements become available, e.g. 'Axisymmetric'.

4.4.2 J2 Plasticity Material

```
nDMaterial J2Plasticity matTag? K? G? sig0? sigInf? delta? H?
```

To construct a J2Plasticity material object with a bulk modulus K, shear modulus G, initial yield stress sig0, final saturation yield stress sigInf, exponential hardening parameter delta and the linear hardening parameter H. The argument matTag is used to uniquely identify this NDMaterial object among NDMaterial objects in the BasicBuilder object.

The material formulations for the J2Plasticity object are 'PlaneStrain2D', 'PlaneStress2D', 'AxiSymmetric2D' and 'ThreeDimensional'. These are the valid strings that can be passed to the continuum elements for the type parameter.

4.4.3 Bidirectional Material

```
nDMaterial Bidirectional matTag? E? sigY? Hiso? Hkin?
```

To construct a Bidirectional material object with an elastic modulus E, yield stress sigY, isotropic hardening modulus Hiso, and kinematic hardening modulus Hkin. A Bidirectional material object is the two dimensional generalization of a one dimensional elastoplasitc model with linear hardening. The argument matTag is used to uniquely identify this NDMaterial object among NDMaterial objects in the BasicBuilder object.

4.5 The section Command

```
section sectionType <specific section args>
```

The section command is used to construct a SectionSD object. SectionSD objects, hereto referred to as Section, represent force-deformation relationships at beam-column sample points. The command has at least one argument, the section type. Currently the following types are permitted: Elastic, Generic1d, GenericNd, Aggregator, and Fiber. The commands for specifying each type are as outlined below.

The valid queries to any section when creating an ElementRecorder are 'force' and 'deformation'.

4.5.1 Elastic Section

```
section Elastic secTag? E? A? Iz? <Iy? G? J?>
```

To construct an ElasticSection object with axial stiffness EA and bending stiffness EIz about the section local z-axis. The values for EIy and GJ, the bending stiffness about the section local y-axis and the section torsional stiffness, respectively, are optional as needed. The argument secTag is used to uniquely identify this Section object among Section objects in the BasicBuilder object.

4.5.2 Generic1d Section

```
section Generic1d secTag? matTag? code
```

To construct a GenericSection1d object which uses a previously defined UniaxialMaterial object, identifed by the argument matTag, to represent a single section force-deformation response quantity. The argument code indicates the force-deformation quantity to be modeled by this section object. Values for code are given in the figure 2. The argument secTag is used to uniquely identify this Section object among Section objects in the BasicBuilder object.

P	Axial force-deformation
Mz	Moment-curvature about section local z-axis
Vy	Shear force-deformation along section local y-axis
My	Moment-curvature about section local y-axis
Vz	Shear force-deformation along section local z-axis
T	Torsion force-deformation

Figure 2: Section force-deformation codes

4.5.3 GenericNd Section

```
section GenericNd secTag? NDTag? code1 code2 ...
```

To construct a GenericSectionNd object which uses a previously defined NDMaterial object, identifed by the argument NDTag, to represent a coupled section force-deformation response quantities. The arguments code1, code2, ... indicate the force-deformation quantities to be modeled by this section object. The number of code arguments must match the order of the NDMaterial object. code1 is mapped to the first stress-strain relation of the NDMaterial, code2 to the second, etc. Values for code are given in Figure 2. The argument secTag is used to uniquely identify this Section object among Section objects in the BasicBuilder object.

4.5.4 Section Aggregator

```
section Aggregator secTag? matTag1? code1 matTag2? code2 ... <-section sectionTag?>
```

To construct a SectionAggregator object which groups previously defined UniaxialMaterial objects, represented by the arguments matTag1 code1 matTag2 code2 ..., into a single section force-deformation model. The optional -section switch is used to specify a previously defined Section object, identified by the argument sectionTag, to which these UniaxialMaterial objects may be added to recursively define a new Section object. The UniaxialMaterial objects aggregated in this Section object are uncoupled from each other as well as from the

Section object represented by sectionTag, if present. Values for code are given in Figure 2. The argument secTag is used to uniquely identify this Section object among Section objects in the BasicBuilder object.

4.5.5 Fiber Section

```
section Fiber secTag? {
   fiber <fiber arguments>
   patch <patch arguments>
   layer <layer arguments>
}
```

To construct a FiberSection object composed of Fiber objects. The available Fiber objects are UniaxialFiber2d/3d, which enforce the Bernoulli beam assumption. The secTag is used to uniquely identify the section object among section objects in the BasicBuilder object. A fiber section has a general geometric configuration formed by subregions of simpler, regular shapes (e.g. quadrilateral, circular and triangular regions) called patches. In addition, layers of reinforcement bars can also be specified. The subcommands patch and layer are used to define the discretization of the section into fibers, and are described below. In these subcommands, the geometric parameters are defined with respect to a planar local coordinate system (y,z). See figures 3 and 7.

The fiber Command

```
fiber yLoc? zLOc? area? matTag?
```

The fiber command is used to construct a fiber object and add it to the section. The arguments consist of the y,z coordinates of the fiber in the section, the area of the fiber and the material tag of the unixial fiber which is used to represent the stress-strain for the area of the fiber.

The patch Command

```
patch patchType <specific patch args>
```

The patch command is used to construct a Patch object. The command has at least one argument, the patch type. Currently the following types are permitted: quad and circ. The fibers generated by the patch commands are UniaxialFiber2d/3d, depending on the dimension of the problem. The commands for specifying each type are described below:

```
patch quad matTag? numSubdivIJ? numSubdivJK? yVertI? zVertI? yVertJ? zVertJ?
                                              yVertK? zVertK? yVertL? zVertL?
```

To construct a patch with a quadrilateral shape. The argument matTag is an integer used to identify the associated UniaxialMaterial (which must be defined previously). The geometry of the patch is defined by four vertices I, J, K and L, as illustrated in figure 3. The arguments numSubdivIJ and numSubdivJK are integers that specify the number of subdivisions (fibers) along the IJ and JK directions, respectively. The last arguments yVertI, zVertI, yVertJ, zVertJ, yVertK, zVertK, yVertL, and zVertL are the coordinates y and z of each of the four vertices specified in sequence (counter-clockwise).

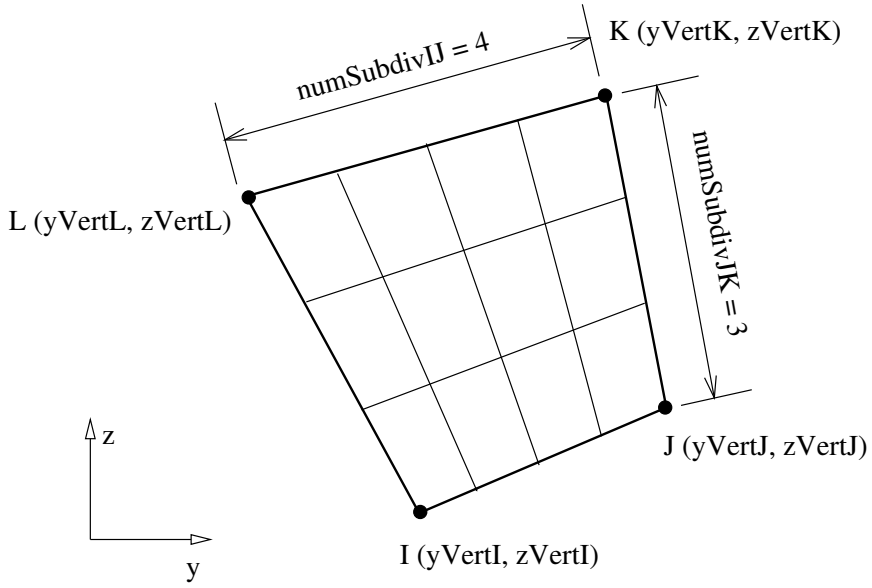

Figure 3: Quadrilateral patch

```
patch circ matTag? numSubdivCirc? numSubdivRad? yCenter? zCenter?
                                  intRad? extRad? startAng? endAng?
```

To construct a patch with a circular shape. The argument matTag is an integer used to identify the associated UniaxialMaterial (which must be defined previously). The arguments numSubdivCirc and numSubdivRad are integers that specify the number of subdivisions (fibers) along the circumferential and radial directions, respectively. The geometry of the patch is defined by its center position (yCenter and zCenter), the internal and external radius (intRad and extRad), and the starting and ending angle (startAng and endAng), according to figure 4.

The layer Command

```
layer layerType <specific patch args>
```

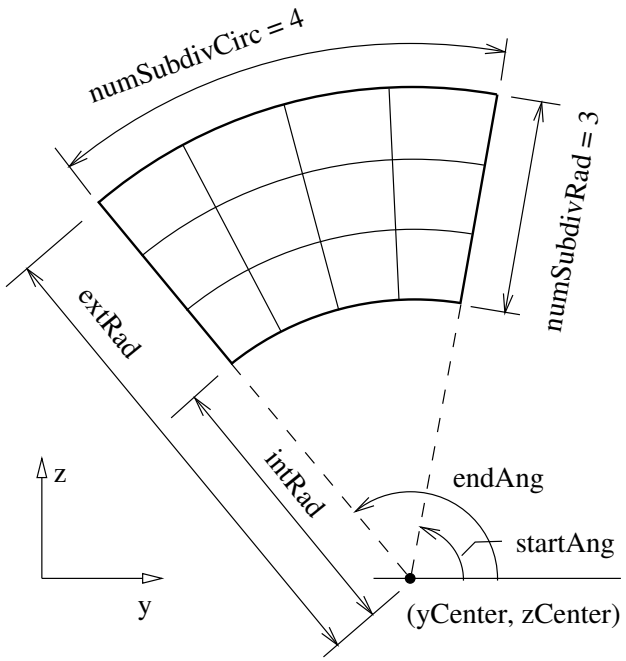

Figure 4: Circular patch

The layer command is used to construct a Layer object. The command has at least one argument, the layer type. Currently the following types are available: straight and circ. The fibers generated by the layer commands are UniaxialFiber2d/3d, depending on the dimension of the problem. The commands for specifying each type are described below:

```
layer straight matTag? numReinfBars? reinfBarArea? yStartPt? zStartPt?
                                                    yEndPt? zEndPt?
```

To construct a straight layer of reinforcing bars. The argument matTag is an integer used to identify the associated UniaxialMaterial (which must be defined previously). The argument numReinfBars is an integer that specifies the number of reinforcing bars, with area reinfBarArea, along the layer. The last arguments yStartPt, zStartPt, yEndPt and zEndPt are the coordinates y and z of the starting and ending points of the reinforcing layer, as represented in figure 5.

```
layer circ matTag? numReinfBars? reinfBarArea?
      yCenter? zCenter? radius? startAng? endAng?
```

To construct a layer with a circular shape. The argument matTag is an integer used to identify the associated material (which must be defined previously). The argument numReinfBars is an integer that specifies the number of reinforcing bars, with area reinfBarArea, along the layer. The geometry of the patch is defined by its center position (yCenter and zCenter), its radius, and the starting and ending angle (startAng and endAng), as shown in figure 6.

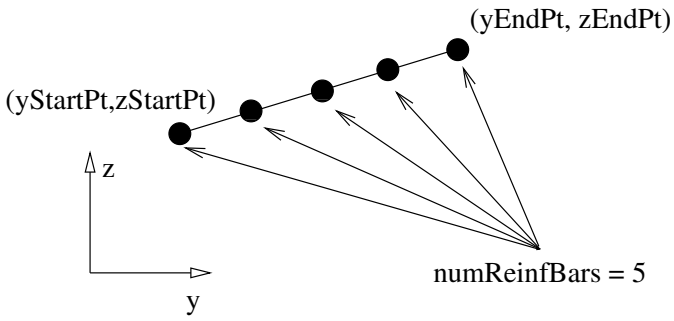

Figure 5: Straight reinforcing layer

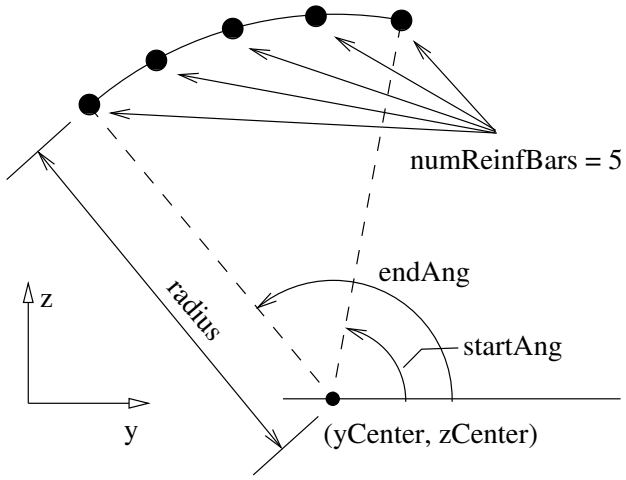

Figure 6: Circular reinforcing layer

4.6 The geomTransf Command

```
geomTransf transfType <specific transf args>
```

The geomTransf command is used to construct a CrdTransf object. A CrdTransf object transforms beam element stiffness and resisting force from the basic system to the global coordinate system. The command has at least one argument, the transformation type. Currently the following types are permitted: Linear and LinearWithPDelta. The commands for specifying each type are as outlined below:

4.6.1 The Linear Transformation

```
geomTransf Linear transfTag? <-jntOffset dXi? dYi? dXj? dYj?>
geomTransf Linear transfTag? vecxzX? vecxzY? vecxzZ?
                  <-jntOffset dXi? dYi? dZi? dXj? dYj? dZj?>
```

To construct a LinearCrdTransf object which performs a linear geometric transformation of beam stiffness and resisting force from the basic system to the global coordinate system. For the three dimensional problem, additional arguments need to be specified. Optional rigid joint offsets can be specified with the -jntOffset switch. The joint offset values dXi, dYi, dZi and dXj, dYj, dZj are absolute offsets with respect to the global coordinate system from element end nodes I and J, respectively. The rigid joint offset arguments depend on the dimension of the current model. The argument transfTag is used to uniquely identify this CrdTransf object among CrdTransf objects in the BasicBuilder object.

The element coordinate system is specified, according to figure 7, as follows: the x axis is the axis connecting the two element nodes; the y and z axis are then defined using a vector that lies on the local xz plane (the components of this vector, vecxzX, vecxzY, vecxzZ are specified with respect to the global coordinate system X,Y,Z). The section is attached to the element such that the (y,z) coordinate system used to specify the section corresponds to the (y,z) axes of the element.

4.6.2 The Linear P-Delta Transformation

```
geomTransf LinearWithPDelta transfTag? <-jntOffset dXi? dYi? dXj? dYj?>
geomTransf LinearWithPDelta transfTag? vecxzX? vecxzY? vecxzZ?
                            <-jntOffset dXi? dYi? dZi? dXj? dYj? dZj?>
```

To construct a LinearCrdTransf object which performs a linear geometric transformation of beam stiffness and resisting force from the basic system to the global coordinate system considering second order P-Delta effects. The arguments vecxzX, vecxzY, and vecxzZ are as defined for the Linear transformation above. Optional rigid joint offsets can be specified with the -jntOffset switch. The joint offset values dXi, dYi, dZi and dXj, dYj, dZj are absolute offsets with respect to the global coordinate system from element end nodes I and J, respectively. The rigid joint offset arguments depend on the dimension of the current model. The argument transfTag is used to uniquely identify this CrdTransf object among CrdTransf objects in the BasicBuilder object.

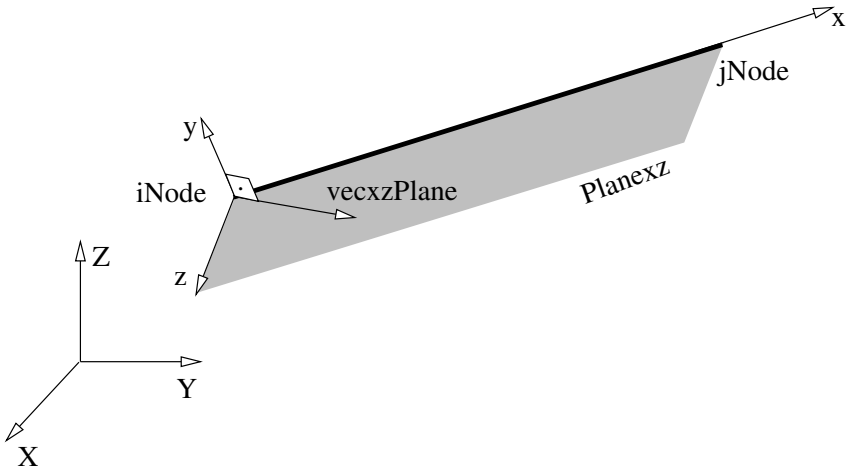

Figure 7: Definition of the local coordinate system

4.7 The element Command

```
element eleType <specific element type args>
```

The element command is used to construct an Element object. The command has at least one argument, the element type. Currently the following types are permitted: truss, elasticBeamColumn, nonlinearBeamColumn, beamWithHinges, zeroLength, zeroLengthSection, zeroLengthND, and quad. The commands for specifying each type are as outlined below:

4.7.1 The Truss Element

```
element truss eleTag? iNode? jNode? A? matTag?
element truss eleTag? iNode? jNode? secTag?
```

There are two ways to construct a truss object. One way is to specify an area and a UniaxialMaterial identifier, the other to specify a Section identifier. To construct a Truss object with an area and a UniaxialMaterial, the analyst specifies the eleTag, which uniquely defines this truss element among all other elements in the domain, the two end nodes, iNode and jNode, the truss bar area A and the tag of the associated UniaxialMaterial, matTag. Note that the material must have already been added to the BasicBuilder object. To construct a Truss object with a Section, the analyst specifies the trussTag, the two end nodes, and the Section tag, secTag.

The valid queries to a truss element when creating an ElementRecorder are 'axialForce', 'stiff', 'material matArg1 matArg2 ..., 'section sectArg1 sectArg2 ...'. There will be more valid queries after the interface for the methods involved have been further developed.

4.7.2 The Elastic Beam Column Element

```
element elasticBeamColumn eleTag? iNode? jNode? A? E? I? transfTag?
```

```
element elasticBeamColumn eleTag? iNode? jNode? A? E? G? Jx? Iy? Iz? transfTag?
```

The arguments to construct an elastic beam-column element depend on the dimension of the problem, ndm. For a two dimensional problem the analyst specifies the eleTag, the two end nodes, the section area, Young's modulus and second moment of section area. For the three dimensional problem the analyst specifies additional material and section properties G, Jx, Iy and Iz. Note that Iz (Iy) is the second moment of area about the beam section local z-axis (y-axis) (Figure 7). For both two and three dimensional problems, the final argument to the elasticBeamColumn command is a transfTag, which identifies a previously defined CrdTransf object.

The valid queries to an elastic beam column element when creating an ElementRecorder are 'stiffness' and 'force'.

4.7.3 The Nonlinear Beam Column Element

```
element nonlinearBeamColumn eleTag? iNode? jNode? numIntgrPts? secTag? transfTag?
                            <-mass massDens> <-iter maxIters tol>
```

The nonlinearBeamColumn element is based on the non-iterative force formulation, and considers the spread of plasticity along the element. The arguments to construct the element are its tag, eleTag, the two end nodes, iNode and jNode, the number of integration points along the element, numIntgrPts, the section tag, secTag (must be pre-defined), and the geometric transformation tag, transfTag (pre-defined). The integration along the element is based on Gauss-Lobatto quadrature rule (two integration points at the element ends). The element is prismatic, i.e. the beam is represented by the section model identified by secTag at each integration point. An element mass density per unit length, massDens, from which a lumped mass matrix is formed, is specified via the -mass switch. An option is also provided for the iterative form of the flexibility formulation by the -iter switch. The arguments for the iterative form are maxIters, the maximum number of iterations to undertake to satisfy element compatibility; and tol, the tolerance for satisfaction of element compatibility. Note that the iterative form can improve the rate of global convergence at the expense of more local element computation.

The valid queries to a nonlinearBeamColumn element when creating an ElementRecorder are 'force', 'stiffness', or 'section secNum secArg1 secArg2 ...'.

4.7.4 The Beam With Hinges Element

```
element beamWithHinges eleTag? iNode? jNode? secTagI? ratioI? secTagJ? ratioJ?
                       E? A? I? transfTag? <-mass massDens> <-iter maxIters tol>
element beamWithHinges eleTag? iNode? jNode? secTagI? ratioI? secTagJ? ratioJ?
                       E? A? Iz? Iy? G? J? transfTag? <-mass massDens>
                       <-iter maxIters tol>
```

The beamWithHinges element is based on the non-iterative flexibility formulation, and considers plasticity to be concentrated over specified hinge lengths at the element ends. The

remaining beam interior is considered to be linear elastic. The arguments to construct the element are its tag, eleTag, the two end nodes, iNode and jNode, the section at node I, secTagI (pre-defined), the ratio of hinge length I to total element length, ratioI, the section at node J, secTagJ (pre-defined), the ratio of hinge length J to total element length, ratioJ, elastic axial stiffness, EA, elastic bending stiffness, EI, and the geometric transformation tag, transfTag (pre-defined). For three-dimensional problems EIz (EIy) is the bending stiffness about the section local z-axis (y-axis) and GJ is the torsional stiffness (Figure 7). Note that these elastic properties are only integrated over the beam interior. The integration over the hinge lengths is mid-point integration. An element mass density per unit length, massDens, from which a lumped mass matrix is formed, is specified via the -mass switch. An option is also provided for the iterative form of the flexibility formulation by the -iter switch. The arguments for the iterative form are maxIters, the maximum number of iterations to undertake to satisfy element compatibility; and tol, the tolerance for satisfaction of element compatibility. Note that the iterative form can improve the rate of global convergence at the expense of more local element computation.

The valid queries to a beamWithHinges element when creating an ElementRecorder are 'force', 'stiffness', 'rotation' (hinge rotation), or 'section secNum secArg1 secArg2 ...'.

4.7.5 The Zero Length Element

```
element zeroLength eleTag? iNode? jNode? -mat matTag1? matTag2? ...
                   -dir dir1? dir2? ... <-orient x1? x2? x3? yp1? yp2? yp3?>
```

Constructs a zero length element defined by two nodes at the same geometric location. The nodes are connected by multiple UniaxialMaterial objects to represent the force-deformation relationship for the element. The zero length element is identified by its tag, eleTag, nodes iNode and jNode, UniaxialMaterial objects (previously defined) identified by matTag1 matTag2 ..., and material directions dir1 dir2 ... . Two optional orientation vectors can be specified for the element. The vector x defines the local x-axis for the element and the vector yp lies in the local x-y plane for the element. The local z-axis is the cross product between x and yp, and the local y-axis is the cross product between the local z-axis and x. If these orientation vectors are not specified, the local element axes coincide with the global axes. The values for dir are 1 through 6, where 1,2,3 indicate translation along the local x,y,z axes respectively; while 4,5,6 indicate rotation about the local x,y,z axes respectively.

The valid queries to a zeroLength element when creating an ElementRecorder are 'force', 'deformation', 'stiff', or 'material matNum matArg1 matArg2 ...'.

4.7.6 The Zero Length Section Element

```
element zeroLengthSection eleTag? iNode? jNode? secTag?
                   -dir dir1? dir2? ... <-orient x1? x2? x3? yp1? yp2? yp3?>
```

Constructs a ZeroLengthSection element defined by two nodes at the same geometric location. The nodes are connected by a single SectionForceDeformation object to represent the

force-deformation relationship for the element. The ZeroLengthSection element is identified by its tag, eleTag, nodes iNode and jNode, and SectionForceDeformation object (previously defined) identified by secTag. Two optional orientation vectors can be specified for the element. The vector x defines the local x-axis for the element and the vector yp lies in the local x-y plane for the element. The local z-axis is the cross product between x and yp, and the local y-axis is the cross product between the local z-axis and x. If these orientation vectors are not specified, the local element axes coincide with the global axes. The section force-deformation response represented by section code P acts along the element local x-axis, and the response for code Vy along the local y-axis. The other modes of section response follow from this orientation.

The valid queries to a ZeroLengthSection element when creating an ElementRecorder are 'force', 'deformation', 'stiff', or 'section secArg1 secArg2 ...'.

4.7.7 The Zero Length ND Element

```
element zeroLengthND eleTag? iNode? jNode? matTag? <uniTag?>
                    -dir dir1? dir2? ... <-orient x1? x2? x3? yp1? yp2? yp3?>
```

Constructs a ZeroLengthND element defined by two nodes at the same geometric location. The nodes are connected by a single NDMaterial object to represent the force-deformation relationship for the element. The ZeroLengthND element is identified by its tag, eleTag, nodes iNode and jNode, and NDMaterial object (previously defined) identified by matTag. Two optional orientation vectors can be specified for the element. The vector x defines the local x-axis for the element and the vector yp lies in the local x-y plane for the element. The local z-axis is the cross product between x and yp, and the local y-axis is the cross product between the local z-axis and x. If these orientation vectors are not specified, the local element axes coincide with the global axes. If the NDMaterial object is of order two, the response lies in the element local x-y plane, and an optional UniaxialMaterial, represented by the argument uniTag, may be used to represent uncoupled behavior orthogonal to this plane, i.e., along the local z-axis. If the NDMaterial is order three, the response is along each of the element local axes. Note that the ZeroLengthND element only represents translational response between its nodes.

The valid queries to a ZeroLengthND element when creating an ElementRecorder are 'force', 'deformation', 'stiff', or 'material matArg1 matArg2 ...'.

4.7.8 The Quad Element

```
element quad eleTag? iNode? jNode? kNode? lNode? thick? type matTag?
             <pressure? rho? b1? b2?>
```

Constructs a FourNodeQuad element which uses the bilinear isoparametric formulation. The quad element is identified by its tag, eleTag, the four nodes iNode, jNode, kNode, and lNode, the element thickness (constant), a string representing the material behavior, type, and an NDMaterial object identified by matTag (previously defined). Valid options for the

parameter type depend on the NDMaterial object and its available material formulations. See the NDMaterial object documentation for valid types. A uniform element normal traction can be specified by the pressure argument. Constant body forces b1 and b2, defined in the isoparametric domain, can be specified as well and are optional, as is the element mass density per unit volume rho, for which a lumped element mass matrix is computed. Consistent nodal loads are computed for the pressure and body forces. The four nodes i through l must be input in counter-clockwise order around the element.

The valid queries to a quad element when creating an ElementRecorder are 'force', 'stiffness', or 'material matNum matArg1 matArg2 ...', where matNum represents the material object at the integration point corresponding to the node numbers in the isoparametric domain.

4.8 The fix Command

```
fix nodeTag? (ndf values?)
```

To construct homogeneous single-point boundary constraints, the user specifies the nodeTag of the node to be constrained and ndf (0,1) values. A 1 specified for the i'th value indicates that the dof for the i'th degree-of-freedom is to be constrained, a 0 that it is to be left unconstrained.

4.9 Types of TimeSeries and Args used in other commands

```
seriesType <arguments for series type>
```

While there is no timeSeries command in the language, a number of commands take as an argument a list of items which defines the TimeSeries object to be constructed as part of the command. The first element of the list is a string identifying the type of TimeSeries object to be constructed. There are a number of of time series objects that can be constructed: Constant, Linear, Rectangular, Sine, and Series.

The arguments for creating the TimeSeries object are as follows:

4.9.1 The Constant Time Series

```
Constant <-factor cFactor?>
```

To associate with the LoadPattern object a TimeSeries object of type ConstantSeries. A Constant causes a load factor of cFactor to be applied to the loads and constraints in the pattern, independent of the time in the domain. The default value of cFactor is 1.0. The analyst has the option of changing this using the optional arguments -factor and cFactor.

4.9.2 The Linear Time Series

```
Linear <-factor cFactor?>
```

To associate with the LoadPattern object a TimeSeries object of type LinearSeries. A LinearSeries causes a load factor of cFactor * time to be applied to the loads and constraints in the pattern. The default value of cFactor is 1.0. The analyst has the option of changing this using the optional arguments -factor and cFactor.

4.9.3 The Rectangular Time Series

```
Rectangular tStart? tFinish? <-factor cFactor?>
```

To associate with the LoadPattern object a TimeSeries object of type RectangularSeries. A RectangularSeries causes a load factor of cFactor * time to be applied to the loads and constraints in the pattern when the value of time is between tStart and tFinish. The default value of cFactor is 1.0. The analyst has the option of changing this using the optional arguments -factor and cFactor.

4.9.4 The Sine Time Series

```
Sine tStart? tFinish? period? <-shift shift?> <-factor cFactor?>
```

To associate with the LoadPattern object a TimeSeries object of type TrigSeries. A TrigSeries is defined by a period, period, and optional phase shift (radians). The TrigSeries causes a load factor of cFactor * $\sin(\frac{2\pi*(time-tStart)}{T} + shift)$ to be applied to the loads and constraints in the pattern when the value of time is between tStart and tFinish. The default value of shift is 0.0 and the default value of cFactor is 1.0. The analyst has the option of changing these values using the optional switches -shift and -factor.

4.9.5 The Path Time Series

```
Series -dt dt? -values {list of points} <-factor cFactor?>
Series -time {list of times}  -values {list of points} <-factor cFactor?>
Series -dt dt? -filePath fileName? <-factor cFactor?>
Series -fileTime fileName1? -filePath fileName2? <-factor cFactor?>
```

To associate with the LoadPattern object a TimeSeries object of type PathSeries or Path-TimeSeries (if time increments not constant). A PathSeries causes a load factor of cFactor * value to be applied to the loads and constraints in the pattern. The value used depends on the time and a linear interpolation between points on the load path. There are a number of ways to specify the load path, for a load path where the points are specified at constant time intervals the '-dt' option. Following dt is either the option '-values' indicating the points are in the accompanying list enclosed in braces, or '-filePath' indicating that the points are contained in the file given by fileName. For a load path where the points are specified at

non-constant time intervals the analyst can provide the time increments and points at these intervals in two strings or two files (not a string and a file). The default value of cFactor is 1.0. The analyst has the option of changing this using the optional arguments -factor and cFactor.

4.10 The pattern Command

```
pattern patternType patternTag? <arguments for pattern type>
```

The pattern command is used to construct a LoadPattern object, its associated TimeSeries object and the Load and Constraint objects for the pattern. There are a number of valid types of patternType: Plain, UniformExcitation, and MultipleSupport.

```
pattern Plain patternTag? {TimeSeriesType and Args} {
     load ...
     sp ...
}
```

The string Plain is used to construct an ordinary LoadPattern object with a unique tag among load patterns in the Domain of patternTag. The third argument is a list which is parsed to construct the TimeSeries object associated with the LoadPattern object. The last argument in the command is a list of commands to create nodal load and single-point constraints.

4.10.1 The UniformExcitation Pattern

```
pattern UniformExcitation patternTag? dir? <-accel {SeriesType and args}>
                                           <-vel0 vel0?>
```

To construct a UniformExcitation load pattern object with a tag, patternTag, unique among all load patterns. The UniformExcitation acts in the direction dir (1,2, or 3) when formulating the inertial loads for the transient analysis. The accelerations used in determining the inertial loads is specified using the -accel flag. The list of arguments after this flag identifies the TimeSeries object to be constructed for the acceleration record. In addition, the user has the option of specifying an initial velocity to be assigned to each node using the -vel0 flag.

4.10.2 The MultipleSupport Pattern

```
pattern MultipleSupport patternTag? {
     groundMotion ...
     imposedMotion ...
}
```

To construct a MultipleSupportExcitation load pattern object with a tag, patternTag, unique among all load patterns. The last argument in the command is a list of commands to create the GroundMotions and ImposedSupportSP constraint objects that are then added to the object to define the multiple support excitation that is being imposed on the model.

4.10.3 The load Command

```
load nodeTag? (ndf values?) <-const> <-pattern patternTag?>
```

The load command is used to construct a NodalLoad object. The first argument to the load command, nodeTag, is an integer tag identifying the node on which the load acts. Following the tag is the ndf reference load values that are to be applied to the node. The nodal load is added to the LoadPattern being defined in the enclosing scope of the pattern command. Optional arguments are the string -const, which identifies the load being applied to the node as being independent of any load factor, and the string -pattern and an integer patternTag identifying the load pattern to which the load is to be added.

4.10.4 The sp Command

```
sp nodeTag? dofTag? value? <-const> <-pattern patternTag?>
```

The sp command is used to construct an SP_Constraint object. The first argument to the sp command, nodeTag, is an integer tag identifying the node on which the single-point constraint acts. Following this tag is an integer, dofTag, identifying the degree-of-freedom at the node being constrained, valid range is 1 through ndf. (Note: fortran indexing at the interpreter, internally OpenSees uses C indexing). The third argument, value, specifies the reference value of the constraint. The constraint is added to the LoadPattern being defined in the enclosing scope of the pattern command. Optional arguments are the string -const, which identifies the constraint being applied to the node as being independent of any load factor, and the string -pattern with an integer patternTag identifying the load pattern to which the constraint is to be added.

4.10.5 The groundMotion Command

```
groundMotion gMotionTag? gMotionType? <type args>
```

The groundMotion command is used to construct a GroundMotion object used by the ImposedMotionSP constraints in a MultipleSuuportExcitation object. The first argument to the groundMotion command, gMotionTag, is an integer tag which uniquely identifies the GroundMotion in the MultipleSupportExcitation. Folowing the tag is a type identifier, identifying the type of GroundMotion object to be constructed, and the specific arguments for each type. At present there are two valid strings for gMotionType: Plain and Interpolated.

The Plain GroundMotion

```
groundMotion gMotionTag? Plain <-accel {SeriesType and Args}>
                               <-vel   {SeriesType and Args}>
                               <-disp  {SeriesType and Args}>
                               <-int   {IntegratorType and Args}>
```

This command is used to construct a plain GroundMotion object. Each GroundMotion object is associated with a number of TimeSeries objects, which define the acceleration record, the velocity record and the displacement record. These are specified using the -accel, -vel and -disp flags and the list argument to define the TimeSeries object. If only the acceleration record is specified, the user has the option of specifying the TimeSeriesIntegrator that is to be used to integrate the acceleration record in order to determine velocity and displacement records. The TimeSeriesIntegrator object is specified using the -int flag and the list argument specifying the TimeSeriesINtegrator object to be constructed.

The Interpolated GroundMotion

```
groundMotion gMotionTag? Interpolated gmTag1? gmTag2? ... -fact fact1? fact2? ...
```

This command is used to construct an InterpoletdGroundMotion. The tags gmTag1, gmTag2, ... identify ground motions which have already been added to the MultipleSupportExcitation. The factors fact1, fact2, ... identify the factors that are used in the interpolation of these ground motions to determine the ground motion for this InterpolatedGroundMotion.

4.10.6 The imposedMotion Command

```
imposedMotion nodeTag? dirn? gMotionTag?
```

This command is used to construct an ImposedMotionSP constraint which is used to enforce the response of a dof, dirn, at a node, nodeTag, in the model. The response enforced at the node at any given time is obtained from the GroundMotion obejct associated with the constraint. This GroundMotion object is determined from the gMotionTag passed in the command. NOTE that the GroundMotion must be added to the MultipleSupportExcitation before the ImposedMotionSP.

4.11 The equalDOF Command

```
equalDOF rNodeTag? cNodeTag? dof1? dof2? ...
```

The equalDOF command is used to construct a multi-point constraint between the nodes identified by rNodeTag and cNodeTag. rNodeTag is the retained, or master node, and cNodeTag is the constrained, or slave node. dof1 dof2 ... represent the nodal degrees of freedom that are constrained at the cNode to be the same as those at the rNode. The valid range for dof1 dof2 ... is 1 through ndf, the number of nodal degrees of freedom.

4.12 The rigidDiaphragm Command

```
rigidDiaphragm perpDirn? masterNodeTag? slaveNodeTag1 ...
```

The rigidDiaphragm command is used to construct a number of MP_Constraint objects. These constraints will constrain certain degrees-of-freedom at the the slave nodes listed to move as if in a rigid plane with the master node. The rigid plane can be the 12, 13 or 23 planes. The rigid plane is specified by the user providing the perpendicular plane direction, ie 3 for 12 plane. The constraint object is constructed assuming small rotations. The rigidDiaphragm command works only for problems in three dimensions with six degrees-of-freedom at the nodes.

4.13 The rigidLink Command

```
rigidLink -type? masterNodeTag? slaveNodeTag
```

The rigidLink command is used to construct a single MP_Constraint object. The type can be either rod or beam. If rod is specified, the translational degrees-of-freedom will be constrained to be exactly the same as those at the master node. If beam is specified, a rigid beam constraint is imposed on the slave node, that is the translational and rotational degrees of freedom are constrained. The constraint object constructed for the beam option assumes small rotations.

5 The analysis Command

```
analysis analysisType
```

The analysis command is used to construct the Analysis object. The valid strings for analysisType are: Static, Transient.

5.1 Static Analysis

```
analysis Static
```

To construct a StaticAnalysis object. The analysis object is constructed with the component objects, i.e. SolutionAlgorithm, StaticIntegrator, ConstraintHandler, DOF_Numberer, LinearSOE, and LinearSolver objects previously created and by the analyst. If none has been created, default objects are constructed and used. These defaults are a NewtonRaphson EquiSolnAlgo with a CTestNormUnbalance with a tolerance of 1e-6 and a maximum of 25 iterations, a PlainHandler ConstraintHandler, an RCM DOF_Numberer, a LoadControl StaticIntegrator with a constant load increment of 1.0, and a profiled symmetric positive definite LinearSOE and LinearSolver.

5.2 Transient Analysis

```
analysis Transient
```

To construct a DircetIntegrationAnalysis object. The analysis object is constructed with the component objects, i.e. SolutionAlgorithm, TransientIntegrator, ConstraintHandler, DOF_Numberer, LinearSOE, and LinearSolver objects previously created by the analyst. If none has been created, default objects are constructed and used. These defaults are a NewtonRaphson EquiSolnAlgo with a CTestNormUnbalance with a tolerance of 1e-6 and a maximum of 25 iterations, a PlainHandler ConstraintHandler, an RCM DOF_Numberer, a Newmark TransientIntegrator with $\gamma = 0.5$ and $\beta = 0.25$, and a profiled symmetric positive definite LinearSOE and LinearSolver.

```
analysis VariableTransient
```

To construct a VariableTimeStepDircetIntegrationAnalysis object. The analysis object is constructed with the component objects, i.e. SolutionAlgorithm, TransientIntegrator, ConstraintHandler, DOF_Numberer, LinearSOE, and LinearSolver objects previously created by the analyst. If none has been created, default objects are constructed and used. These defaults are a NewtonRaphson EquiSolnAlgo with a CTestNormUnbalance with a tolerance of 1e-6 and a maximum of 25 iterations, a PlainHandler ConstraintHandler, an RCM DOF_Numberer, a Newmark TransientIntegrator with $\gamma = 0.5$ and $\beta = 0.25$, and a profiled symmetric positive definite LinearSOE and LinearSolver.

6 The constraints Command

```
constraints constraintHandlerType <args for handler type>
```

The constraints command is used to construct the ConstraintHandler object. The ConstraintHandler object determines how the constraint equations are enforced in the analysis. The valid strings for constraintHandlerType are: Plain, Penalty, Lagrange, and Transformation.

6.1 Plain Constraints

```
constraints Plain
```

This will create a PlainHandler which is only capable of enforcing homogeneous single-point constraints. If other types of constraints exist in the domain, a different constraint handler must be specified.

6.2 Penalty Method

```
constraints Penalty alphaSP? alphaMP?
```

To construct a PenaltyConstraintHandler which will cause the constraints to be enforced using the penalty method. The alphaSP and alphaMP values are the factors used when adding the single-point and multi-point constraints into the system of equations.

6.3 Lagrange Multipliers

```
constraints Lagrange <alphaSP?> <alphaMP?>
```

To construct a LagrangeConstraintHandler which will cause the constraints to be enforced using the method of Lagrange multipliers. The alphaSP and alphaMP values are the factors used when adding the single-point and multi-point constraints into the system of equations. If no values are specified values of 1.0 are assumed. Values other than 1.0 are permited to offset numerical roundoff problems. It should be noted that the resulting system of equations are not positive definite due to the introduction of zeroes on the diagonal by the constraint equations.

6.4 Transformation Method

```
constraints Transformation
```

To construct a TransformationConstraintHandler which will cause the constraints to be enforced using the transformation method. It should be noted that no retained node in an MP_Constraint can also be specified as being a constrained node in another MP_Constraint with the current implementation.

7 The integrator Command

```
integrator integratorType <args for integrator type>
```

The integrator command is used to construct the Integrator object. The Integrator object determines the meaning of the terms in the system of equation object. The valid strings for integratorType for a static analysis are: LoadControl, DisplacementControl, MinUnbalDispNorm, Arclength, and ArcLength1. **It should be noted that static integrators should only be used with a Linear TimeSeries object with a factor of** 1.0. The valid strings for integratorType are for dynamic analysis: Newmark, Newmark1, HHT, and HHT1

7.1 Load Control

```
integrator LoadControl dlambda1? Jd? minLambda? maxLambda?
```

To construct a StaticIntegrator object of type LoadControl. The third argument, dlambda1, is a floating-point number specifying the first load increment (pseudo-time step) in the next invocation of the analysis command. The load increment at subsequent iterations i is related to the load increment at (i-1), dlambda(i-1), and the number of iterations at i-1, J(i-1), by the following: dLambda(i) = dlambda(i-1)*Jd/J(i-1). The floating-point arguments minLambda and maxLambda are used to bound the increment.

7.2 Displacement Control

```
integrator  DisplacementControl nodeTag? dofTag? dU1? Jd? minDU? maxDU?
```

To construct a StaticIntegrator object of type DisplacementControl. The third and fourth arguments, nodeTag and dofTag, are integers identifying the node and the degree-of-freedom at the node (1,ndf), whose response controls the solution. The fourth argument dU1, is a floating-point number specifying the first displacement increment (pseudo-time step) in the next invocation of the analysis command. The displacement increment at subsequent iterations i is related to the displacement increment at (i-1), dU(i-1), and the number of iterations at i-1, J(i-1), by the following: dU(i) = dU(i-1)*Jd/J(i-1). The floating-point arguments minDU and maxDU are used to bound the increment.

7.3 Minimum Unbalanced Displacement Norm

```
integrator MinUnbalDispNorm dlambda11? Jd? minLambda? maxLambda?
```

To construct a StaticIntegrator object of type MinUnbalDispNorm. The third argument, dlambda11, is a floating-point number specifying the first load increment (pseudo-time step) at the first iteration in the next invocation of the analysis command. The first load increment at subsequent iterations i is related to the load increment at (i-1), dlambda(i-1), and the number of iterations at $i-1$, J(i-1), by the following: dLambda(1i) = dlambda(i-1)*Jd/J(i-1). The floating-point arguments minLambda and maxLambda are used to bound the increment.

7.4 Arc-Length Control

```
integrator ArcLength arclength? alpha?
integrator ArcLength1 arclength? alpha?
```

To construct a StaticIntegrator object of type ArcLength or ArcLength1. The third and fourth arguments, are floating-point numbers defining the two arc length parameters to be used.

7.5 Newmark Method

```
integrator Newmark gamma? beta? <alphaM? betaK? betaKinit? betaKcomm?>
integrator Newmark1 gamma? beta? <alphaM? betaK? betaKinit? betaKcomm?>
```

To construct a TransientIntegrator object of type Newmark or Newmark1, (Newmark1 predicts displacement and velocity and sets acceleration to 0, whereas Newmark predicts velocity and acceleration and leaves displacement as is). The third and fourth arguments, are floating-point numbers defining the two Newmark parameters γ and β. Optional arguments are provided for Rayleigh damping, where the damping matrix D = alphaM * M + betaK * Kcurrent + betaKcomm * KlastCommit + betaKinit * Kinit.

7.6 Hilbert-Hughes-Taylor Method

```
integrator HHT alpha? <alphaM? betaK? betaKinit? betaKcomm?>
integrator HHT1 alpha? <alphaM? betaK? betaKinit? betaKcomm?>
```

To construct a TransientIntegrator object of type HHT or HHT1, (HHT1 predicts displacement and velocity and sets acceleration to 0, whereas HHT predicts velocity and acceleration and leaves displacement as is). The third and fourth arguments, are floating-point numbers defining the two Newmark parameters α. Optional arguments are provided for Rayleigh damping, where the damping matrix D = alphaM * M + betaK * Kcurrent + betaKcomm * KlastCommit + betaKinit * Kinit.

8 The algorithm Command

```
algorithm algorithmType <args for algorithm type>
```

The algorithm command is used to construct the Algorithm object. The Algorithm object determines the sequence of steps taken to solve the non-linear equation. The valid strings for algorithmType are: Linear, Newton, ModifiedNewton. None of the present types require additional arguments.

8.1 Linear Algorithm

```
algorithm Linear
```

To construct a Linear algorithm object which takes one iteration to solve the system of equations.

8.2 Newton Algorithm

```
algorithm Newton
```

To construct a NewtonRaphson algorithm object which uses the Newton-Raphson method to advance to the next time step. The tangent is updated at each iteration.

8.3 Newton with Line Search Algorithm

```
algorithm NewtonLineSearch ratio?
```

To construct a NewtonLineSearch algorithm object which uses the Newton-Raphson method with line search to advance to the next time step. If the ratio between the residuals before and after the incremental update is greater than that specified by ratio, which should be betweeen 0.5 and 0.8, the line search algorithm ala Crissfield is employed to try to improve the convergence.

8.4 Modified Newton Algorithm

```
algorithm ModifiedNewton
```

To construct a ModifiedNewton algorithm object which uses the modified Newton-Raphson method to advance to the next time step. The tangent at the first iteration of the current time step is used to iterate to the next time step.

9 The test Command

```
test convergenceTestType <args for test type>
```

Certain SolutionAlgorithm objects require a ConvergenceTest object to determine if convergence has been achieved at the end of an iteration step. The test command is used to construct ConvergenceTest object. The valid strings for convergenceTestType are NormUnbalance, NormDispIncr and EnergyIncr.

9.1 Norm Unbalance Test

```
test NormUnbalance tol? maxNumIter? <printFlag?>
```

To construct a CTestNormUnbalance which tests positive for convergence if the 2-norm of the b vector (the unbalance) in the LinearSOE object is less than tol. A maximum of maxNumIter iterations will be performed before failure to converge will be returned. The optional printFlag can be used to print information on convergence, a 1 will print information on each step, a 2 when convergence has been achieved.

9.2 Norm Displacement Increment Test

```
test NormDispIncr tol? maxNumIter? <printFlag?>
```

To construct a CTestNormDispIncr which tests positive for convergence if the 2-norm of the x vector (the displacement increments) in the LinearSOE object is less than tol. A maximum of maxNumIter iterations will be performed before failure to converge will be returned. The optional printFlag can be used to print information on convergence, a 1 will print information on each step, a 2 when convergence has been achieved.

9.3 Energy Increment Test

```
test EnergyIncr tol? maxNumIter? <printFlag?>
```

To construct a CTestEnergyIncr which tests positive for convergence if the half the inner-product of the x and b vectors (displacement increments and unbalance) in the LinearSOE object is less than tol. A maximum of maxNumIter iterations will be performed before failure to converge will be returned. The optional printFlag can be used to print information on convergence, a 1 will print information on each step, a 2 when convergence has been achieved.

10 The numberer Command

```
numberer numbererType <args for numberer type>
```

The numberer command is used to construct the DOF_Numberer object. The DOF_Numberer object determines the mapping between equation numbers and degrees-of-freedom. The valid strings are: Plain, RCM. None of the present types require additional arguments. As certain system of equation and solver objects do their own mapping, i.e. SuperLU, UmfPack, Kincho's specifying a numberer other than plain may be a waste of time.

11 The system Command

```
system systemType <args for system type>
```

The system command is used to construct the LinearSOE and a LinearSolver objects to store and solve the system of equations in the analysis. The valid types of systemType commands are: BandGeneral, BandSPD, ProfileSPD, SparseGeneral, UmfPack, and SparseSPD.

The BandGeneral SOE

```
system BandGeneral
```

To construct an un-symmetric banded system of equations object which will be factored and solved during the analysis using the Lapack band general solver.

The BandSPD SOE

```
system BandSPD
```

To construct a symmetric positive definite banded system of equations object which will be factored and solved during the analysis using the lapack band spd solver.

The ProfileSPD SOE

```
system ProfileSPD
```

To construct a symmetric positive definite profile system of equations object which will be factored and solved during the analysis using a profile solver.

The SparseGeneral SOE

```
system SparseGeneral <-piv>
```

To construct a general sparse system of equations object which will be factored and solved during the analysis using the SuperLU solver. By default no partial pivoting is performed. The analyst can change this by specifying the -piv option.

The UmfPack SOE

```
system UmfPack
```

To construct a general sparse system of equations object which will be factored and solved during the analysis using the UMFPACK solver.

The SparseSPD SOE

```
system SparseSPD
```

To construct a sparse symmetic positive definite system of equation object which will be factored and solved during the analysis using a sparse solver developed at Stanford by Kincho Law.

12 The recorder Command

```
recorder recorderType <args for type>
```

The recorder command is used to construct a Recorder object. A Recorder object is used to monitor items of interest to the analyst at each commit(). Valid strings for recorderType are MaxNodeDisp, Element Node, display. Note that display is not yet available on the Windows version of opensees, this will be fixed shortly.

12.1 The MaxNodeDisp Recorder

```
recorder MaxNodeDisp dof? node1? node2? ...
```

To construct a recorder of type MaxNodeDisp to record the values of the maximum absolute values of the displacement in the degree-of-freedom direction dof for the nodes node1, node2, ...

12.2 The Node Recorder

```
recorder Node fileName responseType <-time> -node node1? ... -dof dof1? ...
```

To construct a recorder of type NodeRecorder to record the responseType in the degree-of-freedom directions dof1, ..., at the nodes node1, ... The results are saved in the file given by the string fileName. Each line of the file contains the result for a committed state of the domain. An optional argument -time will place the pseudo time of the Domain as the first entry in the line. The responseType defines the response type to be recorded, limited to one of the following: disp, vel, accel and incrDisp for displacements, velocities, accelerations and incremental displacements.

12.3 The Element Recorder

```
recorder Element eleID1? ...  <-file fileName> <-time> arg1? arg2? ...
```

To construct a recorder of type ElementRecorder to record the response at a number of elements with tag eleID1, ... The response recorded is element dependent and depends on the arguments arg1 arg2 ..., which are passed to the setResponse() element method. The results are printed directly to the screen or are saved in a file fileName if the optional -file argument is provided. An optional argument -time will place the pseudo time of the Domain as the first entry in the line. Note on playback, unless the results are stored in a file, nothing will be printed to the secreen for this type of recorder.

12.4 The Display Recorder

```
recorder display windowTitle? xLoc? yLoc? xPixels? yPixels? <-file fineName?>
```

To open a graphical window with the title windowTitle at location xLoc, yLoc which is xPixels wide by yPixels high for the displaying of graphical information. A TclFeViewer object is constructed. This constructor adds a number of additional commands to opensees, similar to the construction of the BasicBuilder. These other commands are used to define the viewing system for the image that is placed on the screen. These commands are under review and will be discussed in the next version of this document. If the optional -file argument is provided, in addition to displaying the model in the window, information is sent to a file so that the images displayed may be redisplayed at a later time.

12.5 The Plot Recorder

```
recorder plot fileName? windowTitle? xLoc? yLoc? xPixels? yPixels?
     <-columns xCol? yCol?>
```

To open a graphical window with the title windowTitle at location xLoc, yLoc which is xPixels wide by yPixels high for the plotting the contents of the file fileName. Unless the optional -columns flag is passed, the first column of the file is used as the x-axis and the second column as the y-axis.

13 The analyze Command

```
analyze numIncr? <dt?> <dtMin?> <dtMax?> <Jd>
```

To invoke analyze(numIncr, <dt>, <dtMin>, <dtMax>, <Jd>) on the Analysis object constructed with the analysis command. Note that dt, the time step increment, is required if a transient analysis or variable time step transient analysis was specified. dtMin, dtMax and Jd (the min step, the max step and the number of iterations the user would like to perform at each time step) are required if a variable time step analysis method is specified. For both transient and variable time step the model will be moved from it's current time to a time ¿= numIncr * dt.

14 The eigen Command

```
eigen numEigenvalues?
```

To perform a generalized eigenvalue problem to determine the first numEigenvalues eigenvalues and eigenvectors. The eigenvectors are stored at the nodes and can be printed out. Currently each invocation of this command constructs a new EigenvalueAnalysis object, each with new component objects: a ConstraintHandler of type Plain, an EigenvalueSOE and solver of type BandArpackSOE and BandArpackSolver and an algorithm of type FrequencyAlgo. These objects are destroyed when the command has finished. This will change.

15 The database Commands

```
database databaseType
```

To construct a FE_Datastore object of type databaseType. Currently there is only one type of datastore object available, that of type FileDatastore. The invocation of this command will add the additional commands **save** and **restore** to the opensees interpreter to allow users to save and restore model states.

```
database File fileName
```

To construct a datastore object of type FileDatastore. The FileDatastore object will save the data into a number of files, e.g fileName.id11 for all ID objects of size 11 that sendSelf() is invoked upon.

15.1 The save Command

```
save commitTag?
```

To save the state of the model in the database. The commitTag is the unique identifier that can be used to restore the state at a latter time.

15.2 The restore Command

```
restore commitTag?
```

To restore the state of the model from the information stored in the database. The state of the model will be the same as when the command save commitTag was invoked.

16 Misc. Commands

The playback Command

```
playback commitTag?
```

To invoke playback on all Recorder objects constructed with the recorder command. The record() method is invoked with the integer commitTag.

The print Command

```
print <fileName>
print <fileName> -node <-flag flag?> <node1? node2? ..>
print <fileName> -ele <-flag flag?> <ele1? ele2? ..>
```

To cause output to be printed to a file or stderr, if fileName is not specified. If no string qualifier is used, the print method is invoked on all objects of the domain. If the string -node or -ele is provided, the print() method is invoked on just the nodes or elements. With the analyst can send the integer flag to the print() method. The analyst can also limit the element and nodes on which the print() method is invoked by supplying the objects tags, ele1, ele2, etc. and node1, node2, etc.

The reset Command

```
reset
```

To set the state of the domain to its original state. Invokes revertToStart() on the Domain object.

The wipe Command

```
wipe
```

To destroy all objects constructed, i.e. to start over again without having to exit and restart the interpreter.

The wipeAnalysis Command

```
wipeAnalysis
```

To destroy all objects constructed for the analysis in order to start a new type of analysis.

The loadConst Command

```
loadConst <-time pseudoTime?>
```

To invoke setLoadConst() on all LoadPattern objectswhich have been created to this point. If the optional string -time or is specified, the pseudo time in the domain will be set to pseudoTime.

The time Command

```
time pseudoTime?
```

To set the pseudo time in the domain to pseudoTime.

The build Command

```
build
```

To invoke build() on the ModelBuilder object. Has no effect on a BasicBuilder object, but will on other types of ModelBuilder objects.

The video Command

```
video -file fileName -window windowName?
```

To construct a TclVideoPlayer object for displaying the images in a file created by the recorder display command. The images are displayed by invoking the command *play*.

17 Example

An example openseesTcl script for the static analysis of the simple linear three bar truss example shown in figure 8 is given in OpenSees/examples/tcl/example1.tcl. First, the analyst creates a ModelBuilder object. In this example a BasicBuilder object is created for a two dimensional problem with two degrees-of-freedom at each node. The construction of the BasicBuilder object adds new commands to the interpreter, i.e. node, material, element, fix and load. It is these commands which can be used by the analyst to construct the model.

```
# create the ModelBuilder object
model BasicBuilder -ndm 2 -ndf 2
```

The analyst then constructs the model. This is done by creating the four Node objects, a Material object, three Element objects, some Constraint objects, and finally a Load Pattern object with a LinearSeries object containing a single NodalLoad object.

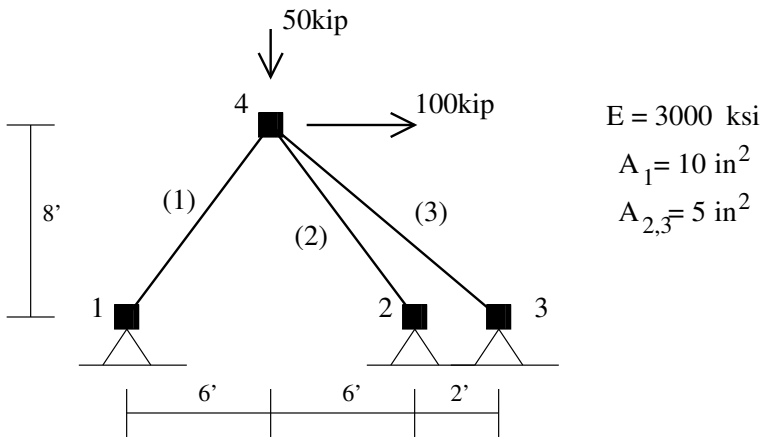

Figure 8: Example 1

```
# node nodeTag xLoc yLoc
node 1   0   0
node 2 144   0
node 3 168   0
node 4  72  96

# material matTag type <type args>
uniaxialMaterial Elastic 1 3000

# truss trussTag iNodeTag jNodeTag Area matTag
element truss 1 1 4 10 1
element truss 2 2 4  5 1
element truss 3 3 4  5 1

# constraint nodeTag xFix? yFix?
fix 1 1 1
fix 2 1 1
fix 3 1 1

# define a load pattern with a load
pattern Plain 1 ‘‘Linear’’ {
  # load nodeTag xForce yForce
  load 4 100 -50
}
```

After the model has been defined, the analyst then constructs the Analysis using commands which already exist in opensees. This is done by first constructing the components of the Analysis object. In this example a BandSPD linear system of equation and a lapack solver (default for BandSPD), a ConstraintHandler object which deals with homogeneous

single point constraints, an Integrator object of type LoadControl with a load step increment of one, an Algorithm object of type Linear, and a DOF_Numberer object of type RCM (reverse Cuthill-Mckee). Once these objects have been created, the Analysis object is constructed. In this case a StaticAnalysis object.

```
# build the components for the analysis object
system BandSPD
constraints Plain
integrator LoadControl 1 1 1 1
algorithm Linear
numberer RCM

# create the analysis object
analysis Static
```

After the Analysis object is constructed a Recorder object is created. In this example we create a NodeRecorder to record the load factor and the two nodal displacements at Node 4, the results are stored in the file /tmp/example.out.

```
# create a Recorder object for the nodal displacements at node 4
recorder Node /tmp/example.out disp -load -node 4 -dof 1 2
```

Finally the analysis is performed and the results are printed.

```
# perform the analysis
analyze 1

# print the results at node 4 and at all elements
print node 4
print ele
playback 1
```

When opensees is run and the commands outlined above are input by the analyst at the interpreter prompt, or are sourced from a file using the source filename command, the following is output by the program.

```
Node: 4
        Coordinates  : 72 96
        commitDisps: 0.530093 -0.177894
        unbalanced Load: 100 -50

Element: 1 type: Truss  iNode: 1 jNode: 4 Area: 10
         strain: 0.00146451 axial load: 43.9352
         unbalanced load: 26.3611 35.1482 -26.3611 -35.1482
         Material: ElasticMaterialModel: 1  E: 3000
```

```
Element: 2 type: Truss  iNode: 2 jNode: 4 Area: 5
         strain: -0.00383642 axial load: -57.5463
         unbalanced load: 34.5278 -46.0371 -34.5278 46.0371
         Material: ElasticMaterialModel: 1  E: 3000

Element: 3 type: Truss  iNode: 3 jNode: 4 Area: 5
         strain: -0.00368743 axial load: -55.3114
         unbalanced load: 39.1111 -39.1111 -39.1111 39.1111
         Material: ElasticMaterialModel: 1  E: 3000

1 0.530093 -0.177894
```