

# XC User's Manual

XC Structural Engineering

September 13, 2024

DRAFT

# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	So...what's XC?	11
1.2	Motivation	11
1.3	XC foundations (on the shoulders of giants)	12
1.3.1	Capabilities	13
1.3.2	Model components	13
1.3.3	Analysis	13
1.4	Open source	13
1.5	Getting started	13
<b>2</b>	<b>Finite element model components</b>	<b>15</b>
2.1	Introduction	15
2.2	Nodes	16
2.2.1	Description	16
2.2.2	Node creation	16
2.2.3	Predefined spaces	16
2.3	Boundary conditions	17
2.3.1	Essential and natural boundary conditions	17
2.3.2	Constraints	17
2.3.2.1	Classification of constraint conditions	17
2.3.2.1.1	Single-freedom constraints.	17
2.3.2.1.2	Multi-freedom constraints	18
2.3.2.2	Methods for imposing the constraints	18
2.3.2.3	Single-point constraints.	18
2.3.2.4	Multi-point constraints	18
2.3.2.4.1	Description	19
<b>3</b>	<b>Materials and sections</b>	<b>21</b>
3.1	Standard uniaxial materials	21
3.1.1	defElasticMaterial	21
3.1.2	defElasticPPMaterial	22
3.1.3	defElastNoTensMaterial	22
3.2	Steel and reinforcing steel materials	23
3.2.1	defCableMaterial	23
3.2.2	defSteel01	23
3.2.3	defSteel02	23
3.2.4	ReinforcingSteel	24

3.3	Concrete materials . . . . .	25
3.3.1	defConcrete01 . . . . .	25
3.4	ND materials . . . . .	26
3.4.1	defElasticIsotropic3d . . . . .	26
3.4.2	defElasticIsotropicPlaneStrain . . . . .	26
3.4.3	defElasticIsotropicPlaneStress . . . . .	27
3.5	Sections . . . . .	27
3.5.1	Sections definition . . . . .	27
3.5.1.1	sectionProperties . . . . .	27
3.5.1.2	defSeccElastica3d . . . . .	28
3.5.1.3	defSeccShElastica3d . . . . .	29
3.5.1.4	defSeccElastica2d . . . . .	29
3.5.1.5	defSeccShElastica2d . . . . .	29
3.5.1.6	CircularSection . . . . .	30
3.5.1.7	RectangularSection . . . . .	31
3.5.1.8	sccRectang . . . . .	31
3.5.1.9	SteelProfile . . . . .	32
3.5.2	Elastic sections . . . . .	33
3.5.2.1	defElasticSection2d . . . . .	33
3.5.2.2	defElasticShearSection2d . . . . .	33
3.5.2.3	defElasticSectionFromMechProp2d . . . . .	34
3.5.2.4	defElasticSection3d . . . . .	34
3.5.2.5	defElasticShearSection3d . . . . .	34
3.5.2.6	defElasticSectionFromMechProp3d . . . . .	35
3.5.2.7	defElasticMembranePlateSection . . . . .	35
3.5.2.8	defElasticPlateSection . . . . .	35
3.5.3	Fiber sections . . . . .	36
3.5.3.1	FiberSet . . . . .	36
3.5.3.2	RCSets . . . . .	36
3.5.3.3	fiberSectionSetupRCSets . . . . .	37
3.5.3.4	createRCFiberSets . . . . .	38
3.5.3.5	resetTensionFibers . . . . .	38
3.5.3.6	fiberSectionSetupRC3Sets . . . . .	38
3.5.3.7	RecordRCColumnSection . . . . .	39
3.5.3.8	RecordShearReinforcement . . . . .	40
3.5.3.9	RecordRCSimpleSection . . . . .	40
3.5.3.10	RecordRCSlabSection . . . . .	41
3.5.3.11	getIMaxPropFiber . . . . .	42
3.5.3.12	getIMinPropFiber . . . . .	43
3.5.3.13	Utils . . . . .	43
3.5.3.13.1	tipoSolicitacion . . . . .	43
3.5.3.13.2	strTipoSolicitacion . . . . .	43
3.5.3.14	gmHorizRowRebars . . . . .	44
3.5.3.15	gmBottomRowRebars . . . . .	44
3.5.3.16	ggmTopRowRebars . . . . .	45
3.5.3.17	gmSquareSection . . . . .	45
3.5.3.18	gmRectangSection . . . . .	45
3.5.3.19	RecordFamMainReinforcement . . . . .	46
3.5.3.20	imprimeMainReinforcement . . . . .	46

3.5.3.21	imprimeArmaduraCortante . . . . .	46
3.5.3.22	informeGeomSeccion . . . . .	47
3.5.4	Sections calculation . . . . .	47
<b>4</b>	<b>Elements</b>	<b>49</b>
4.1	Zero-length elements . . . . .	49
4.1.1	ZeroLength . . . . .	49
4.1.2	ZeroLengthSection . . . . .	52
4.1.3	ZeroLengthContact2D, ZeroLengthContact3D . . . . .	55
4.2	Truss elements . . . . .	58
4.2.1	Truss . . . . .	58
4.2.2	TrussSection . . . . .	62
4.2.3	CorotTruss . . . . .	66
4.2.4	CorotTrussSection . . . . .	70
4.3	Beam-column elements . . . . .	74
4.3.1	ElasticBeam2d . . . . .	74
4.3.2	ElasticBeam3d . . . . .	78
4.3.3	ForceBeamColumn2d . . . . .	83
4.3.4	ForceBeamColumn3d . . . . .	87
4.3.5	Numerical integration options for the forceBeamColumn elements. . . . .	92
<b>5</b>	<b>Loads</b>	<b>93</b>
5.1	Time series . . . . .	93
5.1.1	Constant TimeSeries . . . . .	93
5.1.2	Linear TimeSeries . . . . .	93
5.1.3	Trigonometric TimeSeries . . . . .	93
5.1.4	Triangular TimeSeries . . . . .	93
5.1.5	Rectangular TimeSeries . . . . .	93
5.1.6	Pulse TimeSeries . . . . .	93
5.1.7	Path TimeSeries . . . . .	93
5.1.8	PeerMotion . . . . .	93
5.1.9	PeerNGAMotion . . . . .	93
<b>6</b>	<b>Solution</b>	<b>95</b>
6.1	Analysis and its components . . . . .	95
6.1.1	Constraint handlers . . . . .	95
6.1.1.1	Constraint types . . . . .	95
6.1.1.2	Constraint handler types . . . . .	95
6.1.1.3	Plain Constraints . . . . .	96
6.1.1.4	Lagrange multipliers . . . . .	96
6.1.1.4.1	LagrangeMP_FE . . . . .	96
6.1.1.5	Penalty method . . . . .	97
6.1.1.6	Transformation method . . . . .	97
6.1.2	DOF_Numberer: mapping between equation numbers and degrees of freedom . . . . .	97
6.1.2.1	Plain Numberers . . . . .	98
6.1.2.2	Reverse Cuthill-McKee Numberers . . . . .	98
6.1.3	System of equation and its solution . . . . .	98
6.1.3.1	Band general system of equations . . . . .	98
6.1.3.2	Band symmetric positive definite system of equations . . . . .	98

6.1.3.3	Profile symmetric positive definite system of equations . . . . .	98
6.1.3.4	Sparse general linear system of equations (SuperLU) . . . . .	98
6.1.3.5	Sparse general linear system of equations (UmfPack) . . . . .	98
6.1.3.6	Full general linear system of equations . . . . .	98
6.1.3.7	Sparse symmetric system of equations . . . . .	98
6.1.4	Integrator . . . . .	98
6.1.4.1	Static integrators . . . . .	99
6.1.4.1.1	Load Control . . . . .	99
6.1.4.1.2	Displacement Control . . . . .	99
6.1.4.1.3	Minimum Unbalanced Displacement Norm . . . . .	99
6.1.4.1.4	Arc-Length Control . . . . .	99
6.1.4.2	Transient integrators . . . . .	99
6.1.4.2.1	Central Difference . . . . .	99
6.1.4.2.2	Newmark Method . . . . .	99
6.1.4.2.3	Hilber-Hughes-Taylor Method . . . . .	99
6.1.4.2.4	Generalized Alpha Method . . . . .	99
6.1.4.2.5	TRBDF2 . . . . .	99
6.1.5	Convergence test . . . . .	99
6.1.5.1	Norm Unbalance Test . . . . .	99
6.1.5.2	Norm Displacement Increment Test . . . . .	99
6.1.5.3	Energy Increment Test . . . . .	99
6.1.5.4	Relative Norm Unbalance Test . . . . .	99
6.1.5.5	Relative Norm Displacement Increment Test . . . . .	99
6.1.5.6	Total Relative Norm Displacement Increment Test . . . . .	99
6.1.5.7	Relative Energy Increment Test . . . . .	100
6.1.5.8	Fixed Number of Iterations . . . . .	100
6.1.6	Solution algorithm . . . . .	100
6.1.6.1	Linear Algorithm . . . . .	100
6.1.6.2	Newton Algorithm . . . . .	100
6.1.6.3	Newton with Line Search Algorithm . . . . .	100
6.1.6.4	Modified Newton Algorithm . . . . .	100
6.1.6.5	Krylov-Newton Algorithm . . . . .	100
6.1.6.6	Secant Newton Algorithm . . . . .	100
6.1.6.7	BFGS Algorithm . . . . .	100
6.1.6.8	Broyden Algorithm . . . . .	100
6.1.7	Analyze method . . . . .	100
<b>7</b>	<b>Check routines</b>	<b>103</b>
7.1	Introduction . . . . .	103
7.2	Check routines for steel . . . . .	103
7.2.1	Lateral torsional buckling of steel beams (EC3) . . . . .	103
7.2.1.1	Design lateral torsional buckling resistance $M_{b,Rd}$ . . . . .	103
7.2.1.1.1	Cross section modulus $W_y$ . . . . .	103
7.2.1.1.2	Reduction factor $\chi_{LT}$ . . . . .	104
7.3	Check routines for reinforced concrete . . . . .	106
7.3.1	Verification of RC sections . . . . .	106
7.3.1.1	Sections definition . . . . .	106
7.3.1.2	Limit State at Failure under normal stresses verification . . . . .	106
7.3.1.2.1	lanzaCalculoTNFromXCData . . . . .	106

7.3.1.3	Limit State of Failure due to shear verification . . . . .	108
7.3.1.4	Cracking limit state verification . . . . .	109
7.3.2	Verification of beam sections . . . . .	110
7.3.3	Punching shear calculation . . . . .	110
7.3.3.1	Punching shear calculation according to EC2 . . . . .	110
<b>8</b>	<b>Rough calculations</b>	<b>111</b>
8.1	Punching shear . . . . .	111
8.1.1	Beam deflections . . . . .	111
8.2	Masonry bridge . . . . .	115
8.3	Soil thrust . . . . .	115
<b>A</b>	<b>Generation of combinations to consider in the structural calculation</b>	<b>117</b>
A.1	Introduction . . . . .	117
A.1.1	The Limit States design method . . . . .	117
A.1.2	Design situations . . . . .	117
A.1.3	Actions . . . . .	118
A.1.4	Working life . . . . .	118
A.1.5	Risk level . . . . .	118
A.1.6	Control level . . . . .	119
A.1.7	Combination of actions . . . . .	119
A.1.8	Verification of the structure . . . . .	119
A.2	Actions . . . . .	119
A.2.1	Classification of actions . . . . .	119
A.2.1.1	By their nature . . . . .	120
A.2.1.2	By their variation over time . . . . .	120
A.2.1.3	By their origin . . . . .	120
A.2.1.4	By the structural response which they produce . . . . .	121
A.2.1.5	By their spatial variation . . . . .	121
A.2.1.6	By their relation with other actions . . . . .	121
A.2.1.7	By their participation in a combination . . . . .	121
A.2.2	Values of actions . . . . .	121
A.2.2.1	Characteristic value of an action $F_k$ . . . . .	121
A.2.2.2	Combination value of a variable action $F_{r0}$ . . . . .	121
A.2.2.3	Frequent value of a variable action $F_{r1}$ . . . . .	122
A.2.2.4	Quasi-permanent value of a variable action $F_{r2}$ . . . . .	122
A.2.2.5	Representative value $F_r$ of the actions. Factors of simultaneity . . . . .	122
A.2.2.5.1	Values of $\Psi$ factors of simultaneity . . . . .	122
A.2.2.6	Calculation value $F_d$ of the actions . . . . .	123
A.2.2.6.1	Values of the partial coefficients . . . . .	123
A.3	Design situations . . . . .	126
A.4	Level of quality control . . . . .	126
A.5	Limit states . . . . .	128
A.5.1	Ultimate limit states . . . . .	128
A.5.2	Serviceability limit states . . . . .	128
A.6	Combination of actions . . . . .	129
A.6.1	Combinations of actions for ultimate limit states . . . . .	129
A.6.1.1	Combinations of actions for persistent or transient design situations	130
A.6.1.1.1	Number of combinations to be considered: . . . . .	130

A.6.1.2	Combinations of actions for accidental design situations . . . . .	131
A.6.1.2.1	Number of combinations to be considered: . . . . .	131
A.6.1.3	Combinations of actions for seismic design situations . . . . .	132
A.6.1.3.1	Number of combinations to be considered: . . . . .	132
A.6.2	Combinations of actions for serviceability limit states . . . . .	132
A.6.2.1	Rare combinations: . . . . .	132
A.6.2.2	Frequent combinations: . . . . .	132
A.6.2.3	Quasi-permanent combinations: . . . . .	133
A.6.3	Combinations to be considered in the calculation . . . . .	133
A.6.4	Algorithm to write the complete list of combinations . . . . .	134
A.6.4.1	Combinations for ultimate limit states . . . . .	134
A.6.4.1.1	Combinations of actions for persistent or transient design situations . . . . .	135
A.6.4.1.2	Combinations of actions for accidental design situations . . . . .	135
A.6.4.1.3	Combinations for seismic design situations . . . . .	136
A.6.4.1.4	Calculation algorithm . . . . .	136
A.6.4.2	Combinations for serviceability limit states . . . . .	137
A.6.4.2.1	Calculation algorithm . . . . .	138



# List of Tables

3.65 Sección central en hastial izquierdo. Armadura vertical. (PL\_PF\_OD\_100\_39SecHA1HstICent). 48

A.1	Design working life of the various types of structure (according reference [4]). . .	118
A.2	Recommended values of $\Psi$ factor for climatic actions, according to EHE . . . . .	123
A.3	Recommended values of $\Psi$ factors of simultaneity for climatic loads, according to EHE . . . . .	124
A.4	Recommended values of $\Psi$ factors for buildings, according to EAE . . . . .	124
A.5	Recommended values of $\Psi$ factors of simultaneity, according to EAE . . . . .	125
A.6	Values of $\Psi$ factors of simultaneity according to IAP. . . . .	125
A.7	Partial factor for actions in serviceability limit states according to EHE. . . . .	125
A.8	Partial factor for actions in ultimate limit states according to EHE. . . . .	125
A.9	Partial factor for actions in serviceability limit states according to EAE. . . . .	126
A.10	Partial factor for actions in ultimate limit states according to EAE. . . . .	126
A.11	Partial factor for actions in serviceability limit states according to IAP. . . . .	127
A.12	Partial factor for actions in ultimate limit states according to IAP. . . . .	127

DRAFT

# Chapter 1

## Introduction

### 1.1 So... what's XC?

The “Finite Element Analysis (FEA)” or “Finite Element Method (FEM)” is a numerical method for solving problems of engineering and mathematical physics. This method has proven to be useful to solve problems with one of more of the following characteristics:

- complicated geometries
- complicated load patterns (in space and/or in time).
- complicated material properties.

XC is an open source FEA program designed to solve structural analysis problems. The program can solve various types of problems, from simple linear analysis to complex nonlinear simulations. It has a library of finite elements which allows to model various geometries, and multiple material models, allowing its application in various areas of structural analysis.

### 1.2 Motivation

Someone said that, when the French climber Lionel Terray was asked about his reason to climb a mountain, he simply said “because it’s there”.

Something similar happens with the development of this program. Since I began the study of the finite element method, after studying the analytic solutions to elastic problems (so limited), the scope of their employment to meet structural problems brought on me a great attraction. This, coupled with my love with computer science, made me decide to develop a finite element program that would be useful to calculate structures and that could be modified and expanded in any way the user wanted (a bit optimistic, yeah...). So, first I wrote a Pascal version of the program which could only work with bar-type elements. Afterwards, I wrote a C++ version “from scratch” that was never able to solve any nontrivial problem. Finally I’ve discovered the possibilities offered by the calculation core of *Opensees* and I decided to modify it so it was able to be used in a “engineering office environment” (as opposed to academic use) so to speak. To achieve this objective, the main modifications made to the original code were as follows:

1. Incorporation of some simple algorithms for generation of the finite element mesh. The modeler is able to create structured grids from the description of geometry by means of points, lines, surfaces and solids.

2. Generating graphics using VTK library. This is an open source library for generating graphics for scientific use.
3. At first I've used a macro language, built from the ground, which served to expose all the values from the model (displacements, internal forces, strains, stresses,...) in a way that allows the user to formulate a sentence like "get the ratio between the vertical displacement of the node closest to the center of the beam and the total span of the beam". In 2013 we stopped the development of that custom language and start the migration to Python which was finished in 2015.
4. Utilities for the construction and calculation of design load combinations prescribed by the building codes (EHE, ACI 318, EAE, Eurocodes ...) so as to facilitate the verification of design requirements on each of them.
5. Ability to activate and deactivate elements to enable the analysis of structures built in phases, of geotechnical engineering problems and the strengthening of existing structures.
6. Writing macros to verify the structure and its elements according the criteria prescribed by building codes (axial and bending capacity, shear reinforcement,...).
7. Changing the code so it's linked with "standard" linear algebra libraries (BLAS, Arpack, LAPACK, SuperLU ,...). This eliminates the need to include in the program "ad-hoc" versions of these libraries.
8. Modification of the material models so that support prescribed strains. That makes possible to solve problems involving thermal and rheological actions.
9. Implementation of MP constraints that allows to fix a node to an arbitrary position of an element (the code is based mostly on the article *Modélisation des câbles de précontrainte*. This way we can attach the linear elements, used to model steel cables, to bi-dimensional or three-dimensional elements.

### 1.3 XC foundations (on the shoulders of giants)

The libraries and programs on which XC relies are the following:

- Python: user's programming language (boost.python interface with OpenSees based C++ kernel).
- OpenSees: finite element analysis kernel.
- VTK: visualization routines.
- CGAL: computational geometry algorithms library.
- NumPy: numerical computation.
- SciPy: scientific computing tools for Python.
- LaTeX: document typesetting.
- matplotlib: plotting all kind of math related figures...

### 1.3.1 Capabilities

The main capabilities of the program are:

- Geometry modeling and mesh generation tools.
- OD, 1D, 2D and 3D elements.
- Linear and non-linear analysis, static and dynamic.
- Fiber section models (modelization of RC members,...).
- Activation and deactivation of elements (construction phases,...).
- Tools for implementing structural codes (Eurocodes, ACI, AISC,...) verifications (in progress...).
- Interfaces with Salome and SCIA (in progress...).

XC inherits from OpenSees its advanced capabilities for modeling and analyzing the nonlinear response of systems using a wide range of material models, elements, and solution algorithms. The analysis kernel is designed for parallel computing<sup>1</sup> to allow scalable simulations on high-end computers or for parameter studies.

### 1.3.2 Model components

The program allows the creation of bi or three-dimensional models with 2, 3 or 6 degrees of freedom. The finite element library provides 1D elements (truss and beam-column elements), 2D and 3D elements for continuum mechanics and 2D shell elements.

### 1.3.3 Analysis

Nonlinear analysis requires a wide range of algorithms and solution methods. XC provides nonlinear static and dynamic methods, equation solvers, and methods for handling constraints.

## 1.4 Open source

XC is open-source. This manual provides information about the software architecture, access to the source code, and the development process. The open-source<sup>2</sup> movement allows researchers and users to build upon each others' accomplishments using XC as community-based software.

## 1.5 Getting started

The natural way to get started is to download the software, compile it<sup>3</sup> and run some examples. It goes as follows:

- Primo. Depending on the OS you use you'll need to make some preliminary work. MS Windows and Mac users can take a look here.

---

<sup>1</sup>The Python interface and verifications tests for these capabilities are not yet implemented.

<sup>2</sup>See also *The Cathedral and the Bazaar* by Eric Raymond, which inspired Netscape's decision to release its browser as open-source software.

<sup>3</sup>Yes, for the time being there is no DEB nor RPM binary packages available.

- Secondo. Go to xc repository, clone it and follow the instructions on `install.linux.md`.
- Terzo. You can test some of the examples in `xc_examples` and in `tests`.

You can find some useful links and introductory material [here](#).

DRAFT

## Chapter 2

# Finite element model components

### 2.1 Introduction

XC is comprised of a set of Python modules and objects 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 5 main types of objects, as shown in figure 2.1:

Figure 2.1: Main Objects in an Analysis

Those main objects are:

1. **Preprocessor:** 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 Preprocessor 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 Preprocessor 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 XC 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.
5. **Post-processor:** Postprocessing may be defined as the “art of results representation”. The post-processor is composed by the objects and modules that organize the output of the analysis such that it is easily understandable by the user. It can include checks on the codes and standards to which the construction must comply.

## 2.2 Nodes

### 2.2.1 Description

The nodes of a finite element mesh are the points where the degrees of freedom reside. Each node object has, at least, the following information:

- Coordinates which define its position in space. Typically (x,y,z) coordinates.
- Definition of the degrees of freedom in the node (displacements, rotations,...)

The nodes can also serve to define loads or masses that act over the model at its position.

### 2.2.2 Node creation

To create a node you can use the following commands:

```

odos.newNodeXY(x,y)
odos.newNodeIDXY(tag,x,y)
odos.newNodeXYZ(x,y,z)
odos.newNodeIDXYZ(x,y,z)

```

where:

odos: is a node container obtained from the preprocessor.

tag: is an integer that identifies the node in the model.

(x,y) or (x,y,z): are the cartesian coordinates that define node's position.

### 2.2.3 Predefined spaces

Nodes definition in typical elastic FE models.

```

from model import predefined_spaces as ps
odos= preprocessor.getNodeLoader

```

ps.gdls_elasticidad2D(nodos)	2 node coordinates (x,y)
	2 node DOF ( $u_x, u_y$ )
ps.gdls_resist_materiales2D(nodos)	2 node coordinates (x,y)



<code>ps.gdls_elasticidad3D(nodos)</code>	3 node DOF ( $u_x, u_y, \theta$ )
	3 node coordinates ( $x, y, z$ )
<code>ps.gdls_resist_materiales3D(nodos)</code>	3 node DOF ( $u_x, u_y, u_z$ )
	3 node coordinates ( $x, y$ )
	6 node DOF ( $u_x, u_y, u_z, \theta_x, \theta_y, \theta_z$ )

## 2.3 Boundary conditions

In a finite element problem, the boundary conditions<sup>1</sup> are the specified values of the field variables (displacement, rotations, pore pressures,...).

### 2.3.1 Essential and natural boundary conditions

Essential boundary conditions are conditions that are imposed explicitly on the solution and natural boundary conditions are those that automatically will be satisfied after solution of the problem. Otherwise stated if boundary condition directly involves the nodal freedoms, such as displacements or rotations, it is essential.

This class of boundary conditions involve one or more degrees of freedom and are imposed by manipulating the left hand side (LHS) of the system of equations (the side of the stiffness matrix). The natural boundary conditions are imposed by manipulating the right hand side (RHS) of the system of equations (the side of the force vector). Conditions involving applied loads are natural. This kind of constraints are treated in chapter 5.

### 2.3.2 Constraints

In XC all the classes that represent model constraints inherits from Constraint class

#### 2.3.2.1 Classification of constraint conditions

In the previous description we have said that an essential boundary condition can involve one or more degrees of freedom.

**2.3.2.1.1 Single-freedom constraints.** When there is only one condition involved we call them *single-freedom constraints*. These conditions are mathematically expressible as constraints on individual degrees of freedom:

$$\text{nodal degree of freedom} = \text{prescribed value}$$

For example:

$$u_{x4} = 0, u_{y9} = 0.6 \quad (2.1)$$

These are two single-freedom constraints. The first one is homogeneous while the second one is non-homogeneous.

<sup>1</sup>The following explanation is based on the reference [1].

**2.3.2.1.2 Multi-freedom constraints** The next step up in complexity involves multifreedom equality constraints, or multifreedom constraints for short, the last name being acronymed to MFC. These are functional equations that connect two or more displacement components:

$$\boxed{f(\text{nodal degrees of freedom}) = \text{prescribed value}}$$

or with a more formal mathematical notation:

$$f(u_{x4}, u_{y9}, u_{y109}) = p \quad (2.2)$$

Equation 2.2, in which all displacement components are in the left-hand side, is called the canonical form of the constraint.

An MFC of this form is called *multipoint* or *multinode* if it involves displacement components at different nodes. The constraint is called *linear* if all displacement components appear linearly on the left-hand-side, and *nonlinear* otherwise.

The constraint is called *homogeneous* if, upon transferring all terms that depend on displacement components to the left-hand side, the right-hand side — the “prescribed value” in (8.3) — is zero. It is called *non-homogeneous* otherwise.

### 2.3.2.2 Methods for imposing the constraints

The methods for imposing the constraints are described in 6.1.1.

### 2.3.2.3 Single-point constraints.

A single-point constraint (SPC) enforces a single degree of freedom (normally associated with a node) to a specified value. In structural analysis we use single-point constraints to:

- Constrain or enforce translations and/or rotations of nodes that correspond to structure supports.
- Impose constraints for displacements that correspond to symmetric or antisymmetric boundary conditions.
- Removal of DOFs that correspond to frozen nodes in evolutive problems where some parts of the mesh are deactivated.

### 2.3.2.4 Multi-point constraints

Multipoint constraints are used to impose linear relationships between some of the degrees of freedom of the model as in:

$$\sum_i A_i u_i = 0 \quad (2.3)$$

where  $A_i$  are constant factors and  $u_i$  are degrees of freedom of the model.

This type of constraint allows considerable freedom in describing relations between degrees of freedom. They are used for example to create rigid elements and to link a node to an element.

**2.3.2.4.1 Description** An MP\_Constraint represents a multiple point constraint in the domain. A multiple point constraint imposes a relationship between the displacement for certain dof at two nodes in the model, typically called the *retained* node and the *constrained* node:

$$U_c = C_{cr}U_r \quad (2.4)$$

An MP\_Constraint is responsible for providing information on the relationship between the dof, this is in the form of a constraint matrix,  $C_{cr}$ , and two ID objects, *retainedID* and *constrainedID* indicating the dof's at the nodes represented by  $C_{cr}$ . For example, for the following constraint imposing a relationship between the displacements at node 1, the constrained node, with the displacements at node 2, the retainednode in a problem where the x,y,z components are identified as the 0,1,2 degrees-of-freedom:

$$u_{1,x} = 2u_{2,x} + u_{2,z} \quad (2.5)$$

$$u_{1,y} = 3u_{2,z} \quad (2.6)$$

the constraint matrix is:

$$C_{cr} = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix} \quad (2.7)$$

and the vectors defining the dof's at the nodes are:

$$\text{constrainedID} = [0, 1] \quad (2.8)$$

$$\text{retainedID} = [0, 2] \quad (2.9)$$

DRAFT

## Chapter 3

# Materials and sections

### 3.1 Standard uniaxial materials

#### 3.1.1 defElasticMaterial

Constructs an elastic uniaxial material

```
from materials import typical_materials
typical_materials.defElasticMaterial(preprocessor,name,E)
```

preprocessor  
name  
E

preprocessor name  
name identifying the material  
tangent in the stress-strain diagram (see figure 3.1)

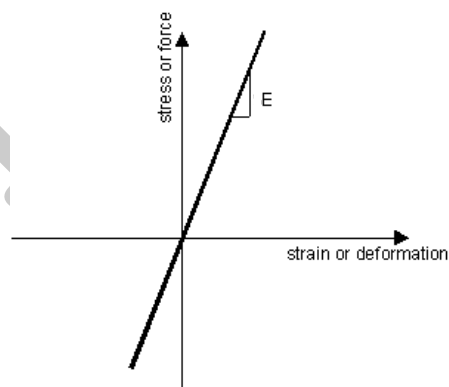


Figure 3.1: Elastic uniaxial material. Stress-strain diagram

### 3.1.2 defElasticPPMaterial

Constructs an elastic perfectly-plastic uniaxial material

```
from materials import typical_materials
typical_materials.defElasticPPMaterial(preprocessor,name,E,fyp,fyn)
```

preprocessor	preprocessor name
name	name identifying the material
E	tangent in the elastic zone of the stress-strain diagram (see figure 3.2)
fyp	stress at which material reaches plastic state in tension (see figure 3.2)
fyn	stress at which material reaches plastic state in compression (see figure 3.2)

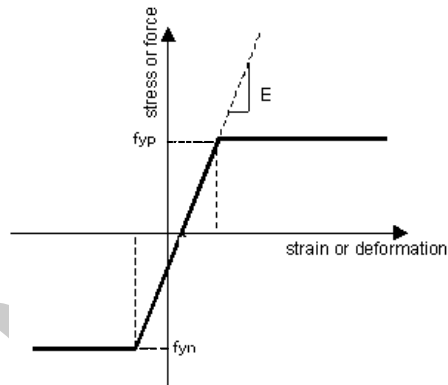


Figure 3.2: Elastic perfectly-plastic uniaxial material. Stress-strain diagram

### 3.1.3 defElastNoTensMaterial

Constructs a uniaxial elastic-no tension material

```
from materials import typical_materials
typical_materials.defElastNoTensMaterial(preprocessor,name,E)
```

preprocessor	preprocessor name
name	name identifying the material
E	tangent in the elastic zone of the stress-strain diagram (see figure 3.3)

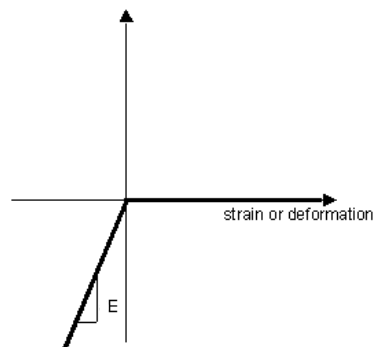


Figure 3.3: Elastic-no tension material. Stress-strain diagram

## 3.2 Steel and reinforcing steel materials

### 3.2.1 defCableMaterial

Constructs a uniaxial bilinear prestressed material. The stress strain ranges from slack (large strain at zero stress) to taught (linear with modulus  $E$ ).

```
from materials import typical_materials
typical_materials.defCableMaterial(preprocessor,name,E,prestress,rho)
```

preprocessor	preprocessor name
name	name identifying the material
E	elastic modulus
prestress	prestress
rho	effective self weight (gravity component of weight per volume transverse to the cable)

### 3.2.2 defSteel01

Constructs a uniaxial bilinear steel material object with kinematic hardening

```
from materials import typical_materials
typical_materials.defSteel01(preprocessor,name,E,fy,b)
```

preprocessor	preprocessor name
name	name identifying the material
E	initial elastic tangent (see figure 3.4)
fy	yield strength (see figure 3.4)
b	strain-hardening ratio: ratio between post-yield tangent and initial elastic tangent (see figure 3.4)

### 3.2.3 defSteel02

Constructs a uniaxial Giuffre-Menegotto-Pinto steel material object with isotropic strain hardening

```
from materials import typical_materials
typical_materials.defSteel02(preprocessor,name,E,fy,b,initialStress)
```

preprocessor	preprocessor name
name	name identifying the material
E	initial elastic tangent (see figure 3.5)

<b>b</b>	strain-hardening ratio: ratio between post-yield tangent and initial elastic tangent)
<b>initialStress</b>	initial stress

The transition from elastic to plastic branches (see figure 3.5) is controlled by parameters R0, R1, R2. The default values R0=15, R1=0.925 and R2=0.15

### 3.2.4 ReinforcingSteel

This class constructs a bilinear stress-strain diagram to carry out the analysis of reinforced concrete according to Eurocode 2. Other national standards, like the spanish EHE and the swiss SIA also adopt this diagram.

#### Parameters

<b>nmbMaterial</b>	name identifying the material
<b>nmbDiagK</b>	name identifying the characteristic stress-strain diagram (default: "dgK"+nmbMaterial)
<b>matTagK</b>	tag of the uniaxial material with the characteristic stress-strain diagram
<b>nmbDiagD</b>	name identifying the design stress-strain diagram (default: "dgD"+nmbMaterial)
<b>matTagD</b>	tag of the uniaxial material with the design stress-strain diagram
<b>fyk</b>	characteristic value of the yield strength
<b>gammaS</b>	partial factor for material (default: 1.15)
<b>Es</b>	elastic modulus of the material (default: 2e11)
<b>emax</b>	maximum strain at failure point
<b>k</b>	ratio between characteristic ultimate stress and characteristic yield stress <sup>(1)</sup> (default: 1.05)

(1): according to annex C of EC2: for class A  $k \geq 1.05$ , for class B  $k \geq 1.08$

#### Methods

<b>fmaxk()</b>	characteristic value of the ultimate strength
<b>fyd()</b>	design value of the yield strength
<b>eyk()</b>	characteristic strain at yield point
<b>eyd()</b>	design strain at yield point
<b>Esh()</b>	post-yield tangent
<b>bsh()</b>	ratio between post-yield tangent and initial elastic tangent
<b>defDiagK(preprocessor)</b>	returns XC uniaxial material (characteristic values)
<b>defDiagD(preprocessor)</b>	returns XC uniaxial material (design values)



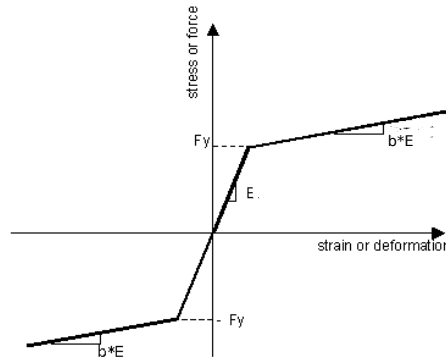


Figure 3.4: Steel001: uniaxial bilinear steel material with kinematic hardening. Stress-strain diagram

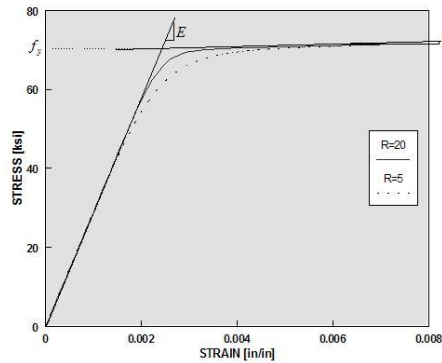


Figure 3.5: Steel002: uniaxial bilinear steel material with isotropic strain hardening. Stress-strain diagram

### 3.3 Concrete materials

#### 3.3.1 defConcrete01

Constructs a uniaxial Kent-Scott-Park concrete material object with degraded linear unloading/reloading stiffness according to the work of Karsan-Jirsa and no tensile strength.

```
from materials import typical_materials
typical_materials.defConcrete01(preprocessor, name, epsc0, fpc, fpcu, epscu)
```

preprocessor	preprocessor name
name	name identifying the material
fpc	concrete compressive strength at 28 days (compression is negative) <sup>(1)</sup>
epsc0	concrete strain at maximum strength (see figure 3.6) <sup>(2)</sup>
fpcu	concrete crushing strength (see figure 3.6)
epscu	concrete strain at crushing strength (see figure 3.6)

- 
- (1): Compressive concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally)
- (2): The initial slope for this model is  $2 * f_{pc}/\epsilon_{psc0}$  (see figure 3.6)

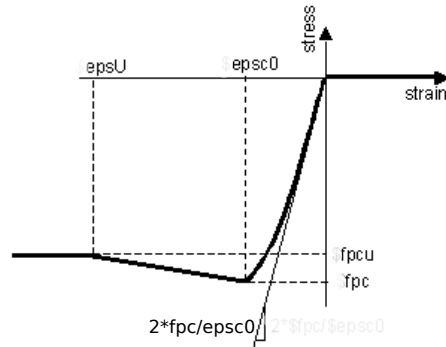


Figure 3.6: Concrete01: uniaxial Kent-Scott-Park concrete material. Stress-strain diagram

### 3.4 ND materials

An ND material is an object that represents the stress-strain relationship at the gauss-point of a continuum element.

#### 3.4.1 defElasticIsotropic3d

Constructs an elastic isotropic material.

```
from materials import typical_materials
typical_materials.defElasticIsotropic3d(preprocessor,name,E,nu,rho)
```

preprocessor	preprocessor name
name	name identifying the material
E	elastic modulus
nu	Poisson's ratio
rho	mass density, optional (default = 0.0)

#### 3.4.2 defElasticIsotropicPlaneStrain

Constructs an elastic isotropic plane-strain material.

```
from materials import typical_materials
typical_materials.defElasticIsotropicPlaneStrain(preprocessor,name,E,nu,rho)
```

<code>preprocessor</code>	preprocessor name
<code>name</code>	name identifying the material
<code>E</code>	elastic modulus
<code>nu</code>	Poisson's ratio
<code>rho</code>	mass density, optional (default = 0.0)

### 3.4.3 `defElasticIsotropicPlaneStress`

Constructs an elastic isotropic plane-stress material.

```
from materials import typical_materials
typical_materials.defElasticIsotropicPlaneStress(preprocessor,name,E,nu,rho)
```

<code>preprocessor</code>	preprocessor name
<code>name</code>	name identifying the material
<code>E</code>	elastic modulus
<code>nu</code>	Poisson's ratio
<code>rho</code>	mass density , optional (default = 0.0)

## 3.5 Sections

A section represents a force-deformation (or resultant stress-strain) relationship at beam-column or plate element.

Three types of sections are going to be considered:

Elastic: defined by material and geometric constants;

Resultant: general nonlinear description of force-deformation response, e.g. moment-curvature;

Fiber: section is discretized into smaller regions for which the material stress-strain response is integrated to give resultant behavior, e.g. reinforced concrete.

### 3.5.1 Sections definition

#### 3.5.1.1 `sectionProperties`

It's an abstract class used to define the properties of a generic section.

```
from materials import sectionProperties
sectionProperties.sectionProperties(name,E,nu)
```

## Parameters

---

<b>name</b>	name identifying the section
<b>E</b>	elastic modulus of material
<b>nu</b>	Poisson's ratio of material

## Abstract methods

---

<b>A()</b>	cross-sectional area of the section (to override)
<b>Iy()</b>	second moment of area about the local y-axis (to override)
<b>Iz()</b>	second moment of area about the local z-axis (to override)
<b>J()</b>	torsional moment of inertia of the section (to override)
<b>G()</b>	transverse modulus of elasticity (defaults to $\frac{E}{2(1+\nu)}$ )
<b>alphaY()</b>	coefficient of distortion about the local y-axis
<b>alphaZ()</b>	coefficient of distortion about the local z-axis
<b>Wyel()</b>	elastic section modulus about the local y-axis
<b>Wzel()</b>	elastic section modulus about the local z-axis

## Methods

---

```

sectionProperties.defSeccElastica3d(preprocessor)
    returns an elastic section appropriate for 3D beam analysis,
    from the data of the section geometric properties
sectionProperties.defSeccShElastica3d(preprocessor)
    returns an elastic section appropriate for 3D beam analysis
    including shear deformations, from the data of the section
    geometric properties
sectionProperties.defSeccElastica2d(preprocessor)
    returns an elastic section appropriate for 2D beam analysis,
    from the data of the section geometric properties
sectionProperties.defSeccShElastica2d(preprocessor)
    returns an elastic section appropriate for 2D beam analysis
    including shear deformations, from the data of the section
    geometric properties

```

### 3.5.1.2 defSeccElastica3d

Returns an elastic section appropriate for 3D beam analysis.

```

from materials import sectionProperties
sectionProperties.defSeccElastica3d(preprocessor,defSecc)

```

<code>preprocessor</code>	preprocessor name
<code>defSecc</code>	name identifying the class object used to define the properties of the section

### 3.5.1.3 `defSeccShElastica3d`

Returns an elastic section appropriate for 3D beam analysis, including shear deformations.

```
from materials import sectionProperties
sectionProperties.defSeccShElastica3d(preprocessor,defSecc)
```

<code>preprocessor</code>	preprocessor name
<code>defSecc</code>	name identifying the class object used to define the properties of the section

### 3.5.1.4 `defSeccElastica2d`

Returns an elastic section appropriate for 2D beam analysis.

```
from materials import sectionProperties
sectionProperties.defSeccElastica2d(preprocessor,defSecc)
```

<code>preprocessor</code>	preprocessor name
<code>defSecc</code>	name identifying the class object used to define the properties of the section

### 3.5.1.5 `defSeccShElastica2d`

Returns an elastic section appropriate for 2D beam analysis, including shear deformations.

```
from materials import sectionProperties
sectionProperties.defSeccShElastica2d(preprocessor,defSecc)
```

<code>preprocessor</code>	preprocessor name
<code>defSecc</code>	name identifying the class object used to define the properties of the section

### 3.5.1.6 CircularSection

This subclass derived from superclass `sectionProperties` is used to define the properties of a circular section.

```
from materials import paramCircularSection
paramCircularSection.CircularSection(name,r,E,nu)
```

#### Parameters

---

<b>name</b>	name identifying the section
<b>r</b>	radius
<b>E</b>	elastic modulus of material
<b>nu</b>	Poisson's ratio of material

#### Methods

---

<b>A()</b>	cross-sectional area of the section
<b>Iy()</b>	second moment of area about the local y-axis
<b>Iz()</b>	second moment of area about the local z-axis
<b>J()</b>	torsional moment of inertia of the section
<b>alphaY()</b>	coefficient of distortion about the local y-axis
<b>alphaZ()</b>	coefficient of distortion about the local z-axis

#### Functions for defining circular sections

---

<b>paramCircularSection.setupSeccCircular(sectionName,r,E,nu)</b>	returns a circular section
<b>paramCircularSection.defSeccCircularElastica3d(preprocessor, defSecc)</b>	returns an elastic circular section appropriate for 3D beam analysis.
<b>paramCircularSection.defSeccCircularShElastica3d(preprocessor, defSecc)</b>	returns an elastic circular section appropriate for 3D beam analysis, including shear deformations.
<b>paramCircularSection.defSeccCircularElastica2d(preprocessor, defSecc)</b>	returns an elastic circular section appropriate for 2D beam analysis.
<b>paramCircularSection.defSeccCircularShElastica2d(preprocessor, defSecc)</b>	returns an elastic circular section appropriate for 2D beam analysis, including shear deformations.
<b>preprocessor</b> : preprocessor name	
<b>defSecc</b> : name identifying an object of the class <code>CircularSection</code> with the properties of the section.	

### 3.5.1.7 RectangularSection

This subclass derived from superclass `sectionProperties` is used to define the properties of a rectangular section.

```
from materials import paramRectangularSection
paramRectangularSection.RectangularSection(,name,b,h,E,nu)
```

#### Parameters

<b>name</b>	name identifying the section
<b>b</b>	width
<b>h</b>	overall depth
<b>E</b>	elastic modulus of material
<b>nu</b>	Poisson's ratio of material

#### Methods

<b>A()</b>	cross-sectional area of the section
<b>Iy()</b>	second moment of area about the local y-axis (Y= weak axis)
<b>Iz()</b>	second moment of area about the local z-axis (Z= strong axis)
<b>J()</b>	torsional moment of inertia of the section
<b>Wye1()</b>	elastic section modulus about the local y-axis
<b>Wze1()</b>	elastic section modulus about the local z-axis
<b>alphaY()</b>	coefficient of distortion about the local y-axis
<b>alphaZ()</b>	coefficient of distortion about the local z-axis

#### Functions for defining parameters of rectangular sections

<b>paramRectangularSection.getJTorsion( b, h)</b>	returns the torsional moment of inertia of the rectangular section
---	--

### 3.5.1.8 sccRectang

This class is used to define a geometric rectangular section.

```
from materials import sccRectg
sccRectg.sccRectang()
```

## Parameters

---

<b>b</b>	width
<b>h</b>	overall depth
<b>nDivIJ</b>	number of cells in IJ (width) direction
<b>nDivJK</b>	number of cells in JK (height) direction

## Methods

---

<b>area()</b>	rectangle area
<b>I1()</b>	second moment of area about the axis through the centre of gravity and parallel to the width dimension (axis 1)
<b>i1()</b>	bending radius of the section with regard to the axis 1
<b>Me1(fy)</b>	yield moment of the rectangular section about the axis 1 fy = yield stress of the section material
<b>S1PosG()</b>	first moment of the area of half a section with regard to the axis 1
<b>Mp1(fy)</b>	plastic moment of the rectangular section about the axis 1 fy = yield stress of the section material
<b>I2()</b>	second moment of area about the axis through the centre of gravity and parallel to the high dimension (axis 2)
<b>i2()</b>	bending radius of the section with regard to the axis 2
<b>Me2(fy)</b>	yield moment of the rectangular section about the axis 2 fy = yield stress of the section material
<b>S2PosG()</b>	first moment of the area of half a section with regard to the axis 2
<b>Mp2(fy)</b>	plastic moment of the rectangular section about the axis 2 fy = yield stress of the section material
<b>discretization(gm,nmbMat)</b>	returns the discretized region gm: name identifying a section geometry nmbMat: name identifying the material

### 3.5.1.9 SteelProfile

This subclass derived from superclass `sectionProperties` is used to define the properties of a structural steel section.

```
from materials import steelProfile
steelProfile.SteelProfile(steel,name,table)
```



## Parameters

---

<b>steel</b>	type of structural steel (e.g. S275JR)
<b>table</b>	file containing a table (dictionary) of steel profiles (e.g. <code>perfiles_metalicos.arcelor.perfiles_he_arcelor</code> )
<b>name</b>	name of the profile in the table (e.g. "HE_400_B")

## Methods

---

<b>A()</b>	cross-sectional area of the section
<b>Iy()</b>	second moment of area about the local y-axis (Y= weak axis)
<b>Iz()</b>	second moment of area about the local z-axis (Z= strong axis)
<b>J()</b>	torsional moment of inertia of the section
<b>EIy()</b>	product of $E_{steel} \times I_y$
<b>EIz()</b>	product of $E_{steel} \times I_z$
<b>GJ()</b>	product of $E_{steel} \times G$
<b>alphaY()</b>	coefficient of distortion about the local y-axis
<b>alphaZ()</b>	coefficient of distortion about the local z-axis

### 3.5.2 Elastic sections

#### 3.5.2.1 defElasticSection2d

Constructs an elastic section appropriate for 2D beam analysis.

```
from materials import typical_materials
typical_materials.defElasticSection2d(preprocessor,name,A,E,I)
```

<b>preprocessor</b>	preprocessor name
<b>name</b>	name identifying the section
<b>A</b>	cross-sectional area of the section
<b>E</b>	Young's modulus of material
<b>I</b>	second moment of area about the local z-axis

#### 3.5.2.2 defElasticShearSection2d

Constructs an elastic section appropriate for 2D beam analysis, including shear deformations.

```
from materials import typical_materials
typical_materials.defElasticShearSection2d(preprocessor,name,A,E,G,I,alpha)
```

<code>preprocessor</code>	preprocessor name
<code>name</code>	name identifying the section
<code>A</code>	cross-sectional area of the section
<code>E</code>	Young's modulus of material
<code>G</code>	shear modulus
<code>I</code>	second moment of area about the local z-axis
<code>alpha</code>	shear shape factor

### 3.5.2.3 `defElasticSectionFromMechProp2d`

Constructs an elastic section appropriate for 2D beam analysis, taking mechanical properties of the section from a `MechProp2d` object.

```
from materials import typical_materials
typical_materials.defElasticSectionFromMechProp2d(preprocessor,name,mechProp2d)
```

<code>preprocessor</code>	preprocessor name
<code>name</code>	name identifying the section
<code>mechProp2d</code>	object that contains mechanical properties of the section

### 3.5.2.4 `defElasticSection3d`

Constructs an elastic section appropriate for 3D beam analysis.

```
from materials import typical_materials
typical_materials.defElasticSection3d(preprocessor,name,A,E,G,Iz,Iy,J)
```

<code>preprocessor</code>	preprocessor name
<code>name</code>	name identifying the section
<code>A</code>	cross-sectional area of the section
<code>E</code>	elastic modulus of material
<code>Iz</code>	second moment of area about the local z-axis
<code>Iy</code>	second moment of area about the local y-axis
<code>J</code>	torsional moment of inertia of the section

### 3.5.2.5 `defElasticShearSection3d`

Constructs an elastic section appropriate for 3D beam analysis, including shear deformations.

```
from materials import typical_materials
typical_materials.defElasticShearSection3d(preprocessor,name,A,E,G,Iz,Iy,J,alpha)
```

<code>preprocessor</code>	preprocessor name
<code>name</code>	name identifying the section
<code>A</code>	cross-sectional area of the section
<code>E</code>	elastic modulus of material
<code>G</code>	transverse modulus of elasticity
<code>Iz</code>	second moment of area about the local z-axis
<code>Iy</code>	second moment of area about the local y-axis
<code>J</code>	torsional moment of inertia of the section
<code>alpha</code>	shear shape factor

### 3.5.2.6 `defElasticSectionFromMechProp3d`

Constructs an elastic section appropriate for 3D beam analysis, taking mechanical properties of the section from a `MechProp3d` object.

```
from materials import typical_materials
typical_materials.defElasticSectionFromMechProp3d(preprocessor, name, mechProp3d)
```

<code>preprocessor</code>	preprocessor name
<code>name</code>	name identifying the section
<code>mechProp3d</code>	object that contains mechanical properties of the section

### 3.5.2.7 `defElasticMembranePlateSection`

Constructs an an isotropic elastic section appropriate for plate and shell analysis.

```
from materials import typical_materials
typical_materials.defElasticMembranePlateSection(preprocessor, name, E, nu, rho, h)
```

<code>preprocessor</code>	preprocessor name
<code>name</code>	name identifying the section
<code>E</code>	elastic modulus
<code>nu</code>	Poisson's ratio
<code>rho</code>	mass density
<code>h</code>	overall depth of section

### 3.5.2.8 `defElasticPlateSection`

Constructs an an isotropic elastic section appropriate for plate analysis.

```
from materials import typical_materials
typical_materials.defElasticPlateSection(preprocessor, name, E, nu, rho, h)
```

preprocessor	preprocessor name
name	name identifying the section
E	elastic modulus
nu	Poisson's ratio
rho	mass density
h	overall depth of section

### 3.5.3 Fiber sections

#### 3.5.3.1 FiberSet

This class constructs a set of all the fibers made of the same material from a fiber section

```
from materials.fiber_section import createFiberSets
createFiberSets.FiberSet(scc,setName,matTag)
```

##### Parameters

scc	name identifying the fiber section
setName	name of the set of fibers to be generated
matTag	tag of the uniaxial material which forms the fibers

##### Methods

getFiberWithMinStrain()	returns the fiber with the minimum strain from the set of fibers
getFiberWithMaxStrain()	returns the fiber with the maximum strain from the set of fibers

#### 3.5.3.2 RCSets

This class constructs both the concrete and reinforced steel fiber sets from a reinforced concrete fiber section.

```
from materials.fiber_section import createFiberSets
createFiberSets.RCSets(scc,concrMatTag, concrSetName,reinfMatTag, reinfSetName)
```

## Parameters

---

<code>scc</code>	name identifying the fiber section
<code>concrMatTag</code>	tag of the uniaxial material that makes up the concrete fibers of the section
<code>concrSetName</code>	name of the set of fibers of concrete to be generated
<code>reinfMatTag</code>	tag of the uniaxial material that makes up the reinforcing steel fibers of the section
<code>reinfSetName</code>	name of the set of fibers of reinforcing steel to be generated

## Methods

---

<code>reselTensionFibers(scc,tensionFibersSetName)</code>	returns a set with those fibers in tension from the total set
<code>getConcreteArea(factor)</code>	returns the cross section area of concrete in the set
<code>getMaxConcreteStrain()</code>	returns the maximum strain in the set of concrete fibers
<code>getConcreteInitialTangent()</code>	returns the initial tangent in the stress-strain diagram of the material that makes up the fibers of concrete
<code>getConcreteCompression()</code>	returns the resultant of compressive stresses in concrete fibers
<code>getNumTensionRebars()</code>	returns the number of reinforcing steel fibers in tension

### 3.5.3.3 fiberSectionSetupRCSets

Returns an object of the class `RCSets`

```
from materials.fiber_section import createFiberSets
createFiberSets.fiberSectionSetupRCSets(scc,concrMatTag, concrSetName,reinfMatTag, reinfSetName)
```

---

<code>scc</code>	name identifying the fiber section
<code>concrMatTag</code>	tag of the uniaxial material that makes up the concrete fibers of the section
<code>concrSetName</code>	name of the set of fibers of concrete to be generated
<code>reinfMatTag</code>	tag of the uniaxial material that makes up the reinforcing steel fibers of the section
<code>reinfSetName</code>	name of the set of fibers of reinforcing steel to be generated

### 3.5.3.4 createRCFiberSets

Constructs the sets of concrete fibers ("hormigon") and reinforcing steel fibers ("reinforcement") for all the elements included in a set of elements.

```
from materials.fiber_section import createFiberSets
createFiberSets.createRCFiberSets(preprocessor, setName, concrMatTag, reinfMatTag)
```

<code>preprocessor</code>	preprocessor name
<code>setName</code>	name identifying the set of elements
<code>concrMatTag</code>	tag of the uniaxial material that makes up the concrete fibers of the section
<code>reinfMatTag</code>	tag of the uniaxial material that makes up the reinforcing steel fibers of the section

### 3.5.3.5 reselTensionFibers

Returns the fibers under tension included in a set of fibers.

```
from materials.fiber_section import createFiberSets
createFiberSets.reselTensionFibers(scc, fiberSetName, tensionFibersSetName)
```

<code>scc</code>	name identifying the fiber section
<code>fiberSetName</code>	name identifying the set of fibers
<code>tensionFibersSetName</code>	name of the set of tensioned fibers returned

### 3.5.3.6 fiberSectionSetupRC3Sets

Returns a set of tensioned fibers ("reinforcementTraccion") of a fiber section of reinforced concrete.

```
from materials.fiber_section import createFiberSets
createFiberSets.fiberSectionSetupRC3Sets(scc, concrMatTag, concrSetName, reinfMatTag, reinfSetName)
```

<code>scc</code>	name identifying the fiber section
<code>concrMatTag</code>	tag of the uniaxial material that makes up the concrete fibers of the section
<code>concrSetName</code>	name of the set of fibers of concrete to be generated
<code>reinfMatTag</code>	tag of the uniaxial material that makes up the reinforcing steel fibers of the section
<code>reinfSetName</code>	name of the set of fibers of reinforcing steel to be generated

### 3.5.3.7 RecordRCCColumnSection

This class is used to define the variables that make up a reinforced concrete section with reinforcement symmetric in both directions (as usual in columns)

```
from materials.fiber_section import defSeccionHAPilar
defSeccionHAPilar.RecordRCCColumnSection()
```

#### Parameters

<code>sectionName</code>	name identifying the section
<code>sectionDescr</code>	section description
<code>gmSectionName</code>	name identifying the geometric section
<code>concrType</code>	type of concrete (e.g. hormigonesEHE.HA25)
<code>concrDiagName</code>	name identifying the characteristic stress-strain diagram of the concrete material
<code>depth</code>	cross-section depth
<code>width</code>	cross-section width
<code>nDivIJ</code>	number of cells in IJ (width) direction
<code>nDivJK</code>	number of cells in JK (height) direction
<code>reinfSteelType</code>	type of reinforcement steel
<code>reinfDiagName</code>	name identifying the characteristic stress-strain diagram of the reinforcing steel material
<code>cover</code>	cover
<code>nRebarsWidth</code>	number of rebars in the width direction of the section (each face)
<code>areaRebarWidth</code>	cross sectional area of each rebar in width direction
<code>nRebarsDepth</code>	number of rebars in the height direction of the section (each face)
<code>areaRebarDepth</code>	cross sectional area of each rebar in height direction
<code>shReinfZ</code>	record of type <code>.defRCSimpleSection.RecordShearReinforcement()</code> defining the shear reinforcement in Z direction
<code>shReinfY</code>	record of type <code>.defRCSimpleSection.RecordShearReinforcement()</code> defining the shear reinforcement in Y direction

#### Methods

```
defGeomRCCColumnSection(matDiagType)
    returns a reinforced concrete section with reinforcement
    symmetric in both directions
    matDiagType ="k" for characteristic diagram, ="d" for design
    diagram
```

### 3.5.3.8 RecordShearReinforcement

This class is used to define the variables that make up a family of shear reinforcing bars.

```
from materials.fiber_section import defRCSimpleSection
defSeccionHASimple.RecordShearReinforcement()
```

#### Parameters

<b>familyName</b>	name identifying the family of shear reinforcing bars
<b>nShReinfBranches</b>	number of effective branches
<b>areaShReinfBranch</b>	area of the shear reinforcing bar
<b>shReinfSpacing</b>	longitudinal distance between transverse reinforcements (defaults to 0.2)
<b>angAlphaShReinf</b>	angle between the shear reinforcing bars and the axis of the member (defaults to $\Pi/2$ )
<b>angThetaConcrStruts</b>	angle between the concrete's compression struts and the axis of the member (defaults to $\Pi/4$ )

#### Methods

<b>getAs()</b>	returns the area per unit length of the family of shear reinforcements
----------------	--

### 3.5.3.9 RecordRCSimpleSection

This class is used to define the variables that make up a rectangular reinforced concrete section with single reinforcement layers in top and bottom faces

```
from materials.fiber_section import defRCSimpleSection
defSeccionHASimple.RecordRCSimpleSection()
```

#### Parameters

<b>sectionName</b>	name identifying the section
<b>sectionDescr</b>	section description
<b>tipoHormigón</b>	type of concrete (e.g. hormigonesEHE.HA25)
<b>concrDiagName</b>	name identifying the characteristic stress-strain diagram of the concrete material
<b>depth</b>	cross-section height
<b>width</b>	cross-section width
<b>nDivIJ</b>	number of cells in IJ (width) direction

continued on next page ...



Parameters ...continued from previous page

---

<code>nDivJK</code>	number of cells in JK (height) direction
<code>reinfSteelType</code>	type of reinforcement steel
<code>reinfDiagName</code>	name identifying the characteristic stress-strain diagram of the reinforcing steel material
<code>nRebarsNeg</code>	number of longitudinal rebars in the negative face of the section
<code>areaRebarNeg</code>	area of each longitudinal rebar in the negative face
<code>rebarsDiamNeg</code>	diameter of the bars in the negative face of the section
<code>coverNeg</code>	cover of longitudinal reinforcement in the negative face
<code>coverLatNeg</code>	lateral cover of longitudinal reinforcement in the negative face
<code>nRebarsPos</code>	number of longitudinal rebars in the positive face of the section
<code>areaRebarPos</code>	area of each longitudinal rebar in the positive face
<code>rebarsDiamPos</code>	diameter of the bars rebar in the positive face of the section
<code>coverPos</code>	cover of longitudinal reinforcement in the positive face
<code>coverLatPos</code>	lateral cover of longitudinal reinforcement in the positive face
<code>coverMin</code>	minimal covering of the longitudinal reinforcement
<code>shReinfZ</code>	record of type <code>.defRCSimpleSection.RecordShearReinforcement()</code> defining the shear reinforcement in Z direction
<code>shReinfY</code>	record of type <code>.defRCSimpleSection.RecordShearReinforcement()</code> defining the shear reinforcement in Y direction

---

Methods

---

```
defGeomRCColumnSection(matDiagType)
    returns a reinforced concrete section with reinforcement
    symmetric in both directions
    matDiagType ="k" for characteristic diagram, ="d" for de-
    sign diagram
```

---

### 3.5.3.10 RecordRCSlabSection

This class is used to define the variables that make up a reinforced concrete slab section with single reinforcement layers in top and bottom faces

```
from materials.fiber_section import.defRCSimpleSection
defSeccionHASimple.RecordRCSlabSection(nmb,desc,depth,concrete,steel,basicCover)
```

## Parameters

---

<b>nmb</b>	basic name identifying the section the name identifying the section in 1 direction is formed adding to <b>nmb</b> the suffix "L"; for the section in 2 direction the suffix added to the basic name is "T"
<b>desc</b>	basic section description
<b>depth</b>	cross-section height (width is considered = 1)
<b>concrete</b>	type of concrete (e.g. hormigonesEHE.HA25)
<b>steel</b>	type of reinforcement steel
<b>basicCover</b>	minimal cover of longitudinal reinforcement

## Methods

---

<b>setMainReinf2neg(diam,area,spacing)</b>	defines the rebar arrangement of the layer of main reinforcement bars corresponding to the negative face of the section in 2 direction <b>diam</b> = diameter of the bar <b>area</b> = area of each bar <b>spacing</b> = space between axis of bars
<b>setMainReinf2pos(diam,area,spacing)</b>	<i>idem</i> for the layer of main reinforcement bars corresponding to the positive face of the section in 2 direction
<b>setMainReinf1neg(diam,area,spacing)</b>	<i>idem</i> for the layer of main reinforcement bars corresponding to the negative face of the section in 1 direction
<b>setMainReinf1pos(diam,area,spacing)</b>	<i>idem</i> for the layer of main reinforcement bars corresponding to the positive face of the section in 1 direction
<b>setShearReinfT(nShReinfBranches,areaShReinfBranch,spacing)</b>	defines the rebar arrangement of the shear reinforcement bars corresponding to the section in 2 direction <b>nShReinfBranches</b> = number of effective branches <b>areaShReinfBranch</b> = area of the shear reinforcing bar <b>spacing</b> = longitudinal distance between shear reinforcements bars
<b>setShearReinfL(nShReinfBranches,areaShReinfBranch,spacing)</b>	<i>idem</i> for shear reinforcement bars corresponding to the section in 1 direction

### 3.5.3.11 getIMaxPropFiber

Returns the fiber in a set with the maximum value of a property.

```
from materials.fiber_section import fiberUtils
fiberUtils.getIMaxPropFiber(fibers,methodName)
```

<code>fibers</code>	name of the set of fibers
<code>methodName</code>	name of the method or property (e.g. "getArea")

### 3.5.3.12 getIMinPropFiber

Returns the fiber with the minimum value of a property from a set of fibers.

```
from materials.fiber_section import fiberUtils
fiberUtils.getIMinPropFiber(fibers,methodName)
```

<code>fibers</code>	name of the set of fibers
<code>methodName</code>	name of the method or property (e.g. "getArea")

### 3.5.3.13 Utils

**3.5.3.13.1 tipoSolicitacion** Returns the following values, depending on the state of stress in the section:

- 1 pure or combined tension where the entire section is under tension;
- 2 pure or combined bending (there are fibers in tension and in compression);
- 3 single or combined compression where all the fibers are in compression.

```
from materials import regimenSeccion
regimenSeccion.tipoSolicitacion(epsCMin, epsSMax)
```

<code>epsCMin</code>	minimum strain in concrete
<code>epsSMax</code>	maximum strain in steel

**3.5.3.13.2 strTipoSolicitacion** Returns:

- "tracción simple o compuesta" in pure or combined tension state;
- "flexotracción" in pure or combined bending state;
- "compresión simple o compuesta" in single or combined compression state;
- "falla" in all other cases.

```
from materials import regimenSeccion
regimenSeccion.strTipoSolicitacion(tipoSol)
```

tipoSol	=1 for pure or combined tension state
	=2 for pure or combined bending state
	=3 for single or combined compression state

#### 3.5.3.14 gmHorizRowRebars

Returns a reinforcement layer parallel to the top and bottom sides of the rectangular section. The row of reinforcement bars is placed at a distance *h* over the horizontal axis of the section.

```
from materials.fiber_section import geomArmaduraSeccionesFibras
geomArmaduraSeccionesFibras.gmHorizRowRebars(sectionGeom, fiberMatName,
nRebars, areaRebar, depth, width, cover, h)
```

sectionGeom	name identifying the geometric section
fiberMatName	name identifying the uniaxial material which forms the fibers
nRebars	number of rebars in the layer
areaRebar	area of each rebar
depth	cross-section height
width	cross-section width
cover	cover
h	distance between the horizontal axis of the cross-section and the reinforcement layer

#### 3.5.3.15 gmBottomRowRebars

Returns a reinforcement layer in the bottom side of a rectangular section.

```
from materials.fiber_section import geomArmaduraSeccionesFibras
geomArmaduraSeccionesFibras.gmBottomRowRebars(sectionGeom, fiberMatName,
nRebars, areaRebar, depth, width, cover)
```

sectionGeom	name identifying the geometric section
fiberMatName	name identifying the uniaxial material which forms the fibers
nRebars	number of rebars in the layer
areaRebar	area of each rebar
depth	cross-section height
width	cross-section width
cover	cover

**3.5.3.16 ggmTopRowRebars**

Returns a reinforcement layer in the top side of a rectangular section.

```
from materials.fiber_section import geomArmaduraSeccionesFibras
geomArmaduraSeccionesFibras.ggmTopRowRebars(sectionGeom, fiberMatName,
nRebars, areaRebar, depth, width, cover)
```

<code>sectionGeom</code>	name identifying the geometric section
<code>fiberMatName</code>	name identifying the uniaxial material which forms the fibers
<code>nRebars</code>	number of rebars in the layer
<code>areaRebar</code>	area of each rebar
<code>depth</code>	cross-section height
<code>width</code>	cross-section width
<code>cover</code>	cover

**3.5.3.17 gmSquareSection**

Returns a square geometric region of fibers' material.

```
from materials.fiber_section import geomSeccionesFibras
geomSeccionesFibras.gmSquareSection(geomSection, fiberMatName, ld, nD)
```

<code>geomSection</code>	name identifying the geometric section
<code>fiberMatName</code>	name identifying the uniaxial material which forms the fibers
<code>ld</code>	side of the square
<code>nD</code>	number of cells in each side of the square

**3.5.3.18 gmRectangSection**

Returns a rectangular geometric region of fibers' material.

```
from materials.fiber_section import geomSeccionesFibras
geomSeccionesFibras.gmRectangSection(geomSection, fiberMatName, h, b,
nDIJ, nDIK)
```

<code>geomSection</code>	name identifying the geometric section
<code>fiberMatName</code>	name identifying the uniaxial material which forms the fibers
<code>h</code>	side of the rectangle in IK direction
<code>b</code>	side of the rectangle in IJ direction
<code>nDIJ</code>	number of cells in IJ direction
<code>nDIK</code>	number of cells in IK direction

### 3.5.3.19 RecordFamMainReinforcement

This class constructs a family of main reinforcing bars.

```
materials.fiber_section import informeGeomSeccion
informeGeomSeccion.RecordFamMainReinforcement(reinfLayer)
```

#### Parameters

---

<b>reinfLayer</b>	name identifying the family of reinforcing bars
<b>nRebars</b>	number of rebars in the layer
<b>rebarsDiam</b>	diameter of the bars
<b>areaRebar</b>	total area of the bars in the family
<b>coverMec</b>	cover of the reinforcement
<b>cdgBarras</b>	position of the reinforcement centre of gravity

---

#### Methods

---

<b>texWrite(archTex)</b>	writes the value of parameters in a file named <b>archTex</b>
--------------------------	---

---

### 3.5.3.20 imprimeMainReinforcement

The function writes in the specified file the characteristics (diameter, area, cover, position of gravity centre, ...) of each main reinforcement family provided in a list of family names.

```
from materials.fiber_section import informeGeomSeccion
informeGeomSeccion.imprimeMainReinforcement(listaFamMainReinforcement,
areaHorm, archTex)
```

---

<b>listaFamMainReinforcement</b>	name identifying a list of names of reinforcement families
<b>areaHorm</b>	area of the concrete section
<b>archTex</b>	name of the output file

---

### 3.5.3.21 imprimeArmaduraCortante

The function writes in the specified file the characteristics (diameter, area, distances, angles, ...) of a family of shear reinforcement bars.

```
from materials.fiber_section import informeGeomSeccion
informeGeomSeccion.imprimeArmaduraCortante(recordShearReinf,
archTex)
```

<code>recordShearReinf</code>	name identifying a family of shear reinforcement
<code>archTex</code>	name of the output file

### 3.5.3.22 informeGeomSeccion

This function prepares a report with the characteristics of a sections, as illustrated in table 3.65

```
from materials.fiber_section import informeGeomSeccion
informeGeomSeccion.informeGeomSeccion(preprocessor,scc, archTex, pathFigura)
```

<code>preprocessor</code>	preprocessor name
<code>scc</code>	name identifying the section
<code>archTex</code>	name of the output file
<code>pathFigura</code>	path to place the figure file representing the reinforcement arrangement

## 3.5.4 Sections calculation

This class is used to perform the stress calculations in a rectangular reinforced concrete section with single reinforcement layers in top and bottom faces.

```
from materials import stressCalc
stressCalc.stressCalc(b,h,r,rp,As,Asp,Ec,Es)
```

### Parameters

<code>b</code>	cross-section width
<code>h</code>	cross-section height
<code>r</code>	cover of longitudinal reinforcement in the positive face
<code>rp</code>	cover of longitudinal reinforcement in the negative face
<code>As</code>	area of longitudinal reinforcement in the positive face
<code>Asp</code>	area of longitudinal reinforcement in the negative face
<code>Ec</code>	concrete elastic modulus
<code>Es</code>	reinforcing steel elastic modulus
<code>N</code>	normal force
<code>M</code>	bending moment

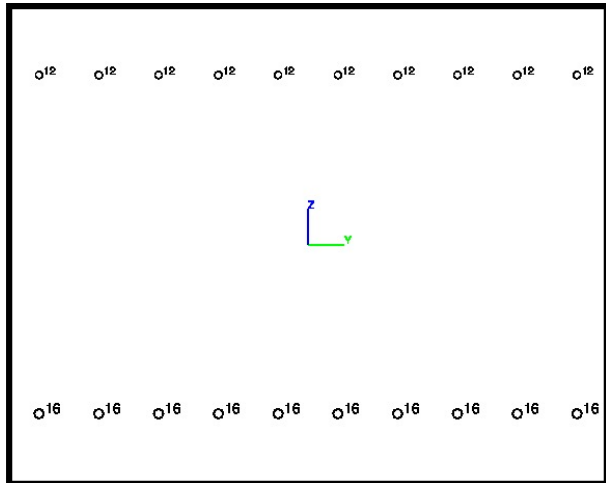
PLPF_OD_100_39SecHA1HstICent									
Sección central en hastial izquierdo. Armadura vertical.									
						width: $b = 1.00 \text{ m}$ depth: $h = 0.70 \text{ m}$			
Materiales:									
Hormigón: HA30		Módulo de deformación longitudinal: $E_c = 28.58 \text{ GPa}$							
Acero: B500S		Módulo elástico: $E_s = 200.00 \text{ GPa}$							
Valores estáticos:									
Sección bruta:									
$A_{bruta} = 0.700 \text{ m}^2$		Tensor de inercia ( $\text{cm}^4$ ): $\begin{pmatrix} 647.78 & 0.00 & 0.00 \\ 0.00 & 285.83 & -0.00 \\ 0.00 & -0.00 & 583.33 \end{pmatrix}$							
C.D.G.: $(0.00, 0.00) \text{ m}$									
Sección homogeneizada:									
$A_{homog.} = 0.737 \text{ m}^2$		Tensor de inercia ( $\text{cm}^4$ ): $\begin{pmatrix} 647.78 & 0.00 & 0.00 \\ 0.00 & 315.25 & -0.00 \\ 0.00 & -0.00 & 613.81 \end{pmatrix}$							
C.D.G.: $(0.00, -0.00) \text{ m}$									
Armadura pasiva:									
Área total $A_s = 31.40 \text{ cm}^2$ Cuantía geométrica $\rho = 4.49\%$									
Familias de reinforcement principal:									
Id	n. barras	$\phi$ (mm)	área ( $\text{cm}^2$ )	c. geom. (‰)	cover. mec. (cm)	$y_{cdg}$ (m)	$z_{cdg}$ (m)		
neg	10	16	20.10	2.87	5.0	0.000	-0.282		
pos	10	12	11.30	1.61	5.0	0.000	0.284		
Familias de reinforcement de cortante:									
Id	n. ramas	$\phi$ (mm)	área ( $\text{cm}^2$ )	sep. (cm)	area/m ( $\text{cm}^2/\text{m}$ )	$\alpha$ (°)	$\beta$ (°)		
Vz	0	0	0.00	20.0	0.00	90.0	45.0		
Vy	0	0	0.00	20.0	0.00	90.0	45.0		

Table 3.65: Sección central en hastial izquierdo. Armadura vertical. (PLPF\_OD\_100\_39SecHA1HstICent).



# Chapter 4

## Elements

### 4.1 Zero-length elements

#### 4.1.1 ZeroLength

The ZeroLength class represents an element defined by two nodes at the same geometric location, hence it has zero length.

The nodes are connected by means of uniaxial materials to represent the force-deformation relationship for the element.

ZeroLength elements are constructed with a *tag* in a domain of *dimension* 1, 2, or 3, connected by nodes *Nd1* and *Nd2*. The vector  $\vec{x}$  defines the local x-axis for the element and the vector  $\vec{y}\vec{p}$  lies in the local x-y plane for the element. The local z-axis is the cross product between  $\vec{x}$  and  $\vec{y}\vec{p}$ , and the local y-axis is the cross product between the local z-axis and  $\vec{x}$ .

The force-deformation relationship for the element is given by a pointer *theMaterial* to a **UniaxialMaterial** model acting in local *direction*.

The local *direction* is 0, 1, 2 for translation in the local x, y, z axes or 3, 4, 5 for rotation about the local x, y, z axes.

```
preprocessor=xc.ProblemaEF().getModelador
ZeroLengthElement=preprocessor.getElementLoader.newElement("zero_length",
xc.ID([Nd1Tag,Nd2Tag]))
```

Nd1Tag,Nd2Tag

tags of the nodes connected by the element

#### Parameters

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getIVector</code>	vector in the element local x-axis direction

continued on next page ...

Parameters ...continued from previous page

---

<code>getJVector</code>	vector in the element local y-axis direction
<code>getKVector</code>	vector in the element local z-axis direction

## Methods

---

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not
<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes
<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation
<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	returns the element centroid coordinates <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions <code>getLongTributaria True</code> to consider the initial geometry shape, <i>False</i> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code>	

---

continued on next page ...

Methods ...continued from previous page

---

<code>calculaLongsTributarias(geomInicial)</code>	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument
<code>getLongTributariaByTag(tag)</code>	returns the tributary length associated with the node labelled with the <code>tag</code> given as argument
<code>calculaAreasTributarias(geomInicial)</code>	returns the tributary area associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getAreaTributaria(Node)</code>	returns the tributary area associated with the <code>Node</code> given as argument
<code>getAreaTributariaByTag(tag)</code>	returns the tributary area associated with the node labelled with the <code>tag</code> given as argument
<code>calculaVolsTributarios(geomInicial)</code>	returns the tributary volume associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getVolTributario(Node)</code>	returns the tributary volume associated with the <code>Node</code> given as argument
<code>getVolTributarioByTag(tag)</code>	returns the tributary volume associated with the node labelled with the <code>tag</code> given as argument
<code>getMaxCooNod(axisIdx)</code>	returns the maximum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMinCooNod(axisIdx)</code>	returns the minimum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>setupVectors(x,yp)</code>	establish orientation of element for the transformation matrix $\vec{x}$ vector in global coordinates defining local x-axis $\vec{y}_p$ vector in global coordinates which lies in the local x-y plane for the element local z-axis is defined by the vector $\vec{z} = \vec{x} \times \vec{y}_p$

---

continued on next page ...

Methods ...continued from previous page

---

<code>clearMaterials()</code>	clears all material definition of the element
<code>setMaterial(dir,matName)</code>	assigns uni-axial materials to the different directions of the element <code>dir</code> : integer representing the direction in which the uni-axial material acts: <code>dir</code> is 0,1,2 for translation in the local x, y, z axes or 3, 4, 5 for rotation about the local x, y, z axes. <code>matName</code> : string representing the name of the material
<code>getMaterials</code>	

#### 4.1.2 ZeroLengthSection

The `ZeroLengthSection` class represents an element defined by two nodes at the same geometric location, hence it has zero length.

The nodes are connected by a `SectionForceDeformation` object which represents the force-deformation relationship for the element.

`ZeroLength` elements are constructed with a *tag* in a domain of *dimension* 1, 2, or 3, connected by nodes *Nd1* and *Nd2*. The vector  $\vec{x}$  defines the local x-axis for the element and the vector  $\vec{y}\vec{p}$  lies in the local x-y plane for the element. The local z-axis is the cross product between  $\vec{x}$  and  $\vec{y}\vec{p}$ , and the local y-axis is the cross product between the local z-axis and  $\vec{x}$ .

The force-deformation relationship for the element is obtained by invoking `getCopy()` on the **SectionForceDeformation** pointer *theSection*. The section model acts in the local space defined by the  $\vec{x}$  and  $\vec{y}\vec{p}$  vectors. The section axial force-deformation acts along the element local x-axis and the section y-z axes directly corresponds to the local element y-z axes.

```
preprocessor=xc.ProblemaEF().getModelador
ZeroLengthElement=preprocessor.getElementLoader.newElement(
"zero_length_section",xc.ID([Nd1Tag,Nd2Tag]))
```

`Nd1Tag,Nd2Tag` tags of the nodes connected by the element

#### Parameters

---

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getIVector</code>	vector in the element local x-axis direction
<code>getJVector</code>	vector in the element local y-axis direction

---

continued on next page ...

Parameters ...continued from previous page

---

`getKVector`            vector in the element local z-axis direction

## Methods

---

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not
<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes
<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation
<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	returns the element centroid coordinates <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions <code>getLongTributaria True</code> to consider the initial geometry shape, <i>False</i> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code>	
<code>calculaLongsTributarias(geomInicial)</code>	

---

continued on next page ...

Methods ...continued from previous page

---

	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument
<code>getLongTributariaByTag(tag)</code>	returns the tributary length associated with the node labelled with the <code>tag</code> given as argument
<code>calculaAreasTributarias(geomInicial)</code>	returns the tributary area associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getAreaTributaria(Node)</code>	returns the tributary area associated with the <code>Node</code> given as argument
<code>getAreaTributariaByTag(tag)</code>	returns the tributary area associated with the node labelled with the <code>tag</code> given as argument
<code>calculaVolsTributarios(geomInicial)</code>	returns the tributary volume associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getVolTributario(Node)</code>	returns the tributary volume associated with the <code>Node</code> given as argument
<code>getVolTributarioByTag(tag)</code>	returns the tributary volume associated with the node labelled with the <code>tag</code> given as argument
<code>getMaxCooNod(axisIdx)</code>	returns the maximum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMinCooNod(axisIdx)</code>	returns the minimum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>setupVectors(x,yp)</code>	establish orientation of element for the transformation matrix <code>x</code> vector in global coordinates defining local x-axis <code>yp</code> vector in global coordinates which lies in the local x-y plane for the element local z-axis is defined by the vector $\vec{z} = \vec{x} \times \vec{yp}$
<code>getSection()</code>	returns the section axial force-deformation associated with the element

---

continued on next page ...

Methods ...continued from previous page

<code>getMaterial()</code>	returns the section axial force-deformation associated with the element
----------------------------	---

### 4.1.3 ZeroLengthContact2D, ZeroLengthContact3D

These classes are used to constructs a zeroLengthContact2D element or a zeroLengthContact3D element, which are Node-to-node frictional contact element used in two dimensional analysis and three dimensional analysis.

The contact element is node-to-node contact. Contact occurs between two contact nodes when they come close. The relation follows Mohr-coulomb law:  $T = \mu \cdot N + c$ , where  $T$  is tangential force and  $N$  is normal force across the interface;  $\mu$  is friction coefficient and  $c$  is total cohesion (summed over the effective area of contact nodes).

The contact node pair in node-to-node contact element is termed ■master node■ and ■slave node■, respectively. Master/slave plane is the contact plane which the master/slave node belongs to. The discrimination is made solely for contact detection purpose. User need to specify the corresponding out normal of the master plane, and this direction is assumed to be unchanged during analysis. For simplicity, 3D contact only allows 3 options to specify the directions of the contact plane. The convention is: out normal of master plane always points to positive axial direction (+X or +Y, or +Z)

For 2D contact, slave nodes and master nodes must be 2 DOF. For 3D contact, slave nodes and master nodes must be 3 DOF.

The resulted tangent from the contact element is non-symmetric. Switch to non-symmetric matrix solver.

```
preprocessor=xc.ProblemaEF().getModelador
ZeroLengthElement=preprocessor.getElementLoader.newElement(
"zero_length_contact_2d",xc.ID([Nd1Tag,Nd2Tag]))
"zero_length_contact_3d",xc.ID([Nd1Tag,Nd2Tag]))
```

Nd1Tag,Nd2Tag                      tags of master and slave nodes

#### Parameters

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getIVector</code>	vector in the element local x-axis direction
<code>getJVector</code>	vector in the element local y-axis direction
<code>getKVector</code>	vector in the element local z-axis direction

continued on next page ...

Parameters ...continued from previous page

## Methods

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not
<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes
<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation
<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	returns the element centroid coordinates <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions <code>getLongTributaria True</code> to consider the initial geometry shape, <i>False</i> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code>	
<code>calculaLongsTributarias(geomInicial)</code>	

continued on next page ...



Methods ...continued from previous page

	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument
<code>getLongTributariaByTag(tag)</code>	returns the tributary length associated with the node labelled with the <code>tag</code> given as argument
<code>calculaAreasTributarias(geomInicial)</code>	returns the tributary area associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getAreaTributaria(Node)</code>	returns the tributary area associated with the <code>Node</code> given as argument
<code>getAreaTributariaByTag(tag)</code>	returns the tributary area associated with the node labelled with the <code>tag</code> given as argument
<code>calculaVolsTributarios(geomInicial)</code>	returns the tributary volume associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getVolTributario(Node)</code>	returns the tributary volume associated with the <code>Node</code> given as argument
<code>getVolTributarioByTag(tag)</code>	returns the tributary volume associated with the node labelled with the <code>tag</code> given as argument
<code>getMaxCooNod(axisIdx)</code>	returns the maximum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMinCooNod(axisIdx)</code>	returns the minimum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>setupVectors(x,yp)</code>	establish orientation of element for the transformation matrix $\vec{x}$ vector in global coordinates defining local x-axis $\vec{yp}$ vector in global coordinates which lies in the local x-y plane for the element local z-axis is defined by the vector $\vec{z} = \vec{x} \times \vec{yp}$

## 4.2 Truss elements

### 4.2.1 Truss

This class is used to constructs a truss element object defined by two nodes connected by means of a previously defined uniaxial material. The truss element does not include geometric nonlinearities, even when used with beam-columns utilizing P-Delta or Corotational transformations. The truss element considers strain-rate effects, and is thus suitable for use as a damping element.

```
preprocessor=xc.ProblemaEF().getModelador
trussElement=preprocessor.getElementLoader.newElement(
"truss",xc.ID([Nd1Tag,Nd2Tag]))
```

Nd1Tag,Nd2Tag                      tags of the nodes connected by the element

#### Parameters

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getCoordTransf()</code>	returns the identifier of the coordinate-transformation associated with the element

#### Methods

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not
<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes
<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation

continued on next page ...

Methods ...continued from previous page

---

<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	returns the element centroid coordinates <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions <code>geomInicial = True</code> to consider the initial geometry shape, <code>False</code> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code>	
<code>calculaLongsTributarias(geomInicial)</code>	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument
<code>getLongTributariaByTag(tag)</code>	returns the tributary length associated with the node labelled with the <code>tag</code> given as argument
<code>calculaAreasTributarias(geomInicial)</code>	returns the tributary area associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getAreaTributaria(Node)</code>	returns the tributary area associated with the <code>Node</code> given as argument
<code>getAreaTributariaByTag(tag)</code>	returns the tributary area associated with the node labelled with the <code>tag</code> given as argument
<code>calculaVolsTributarios(geomInicial)</code>	

---

continued on next page ...

Methods ...continued from previous page

---

	returns the tributary volume associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getVolTributario(Node)</code>	returns the tributary volume associated with the <code>Node</code> given as argument
<code>getVolTributarioByTag(tag)</code>	returns the tributary volume associated with the node labelled with the <code>tag</code> given as argument
<code>getMaxCooNod(axisIdx)</code>	returns the maximum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMinCooNod(axisIdx)</code>	returns the minimum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMEDCellType()</code>	interface with MED data format for Salome
<code>vector2dUniformLoadGlobal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector $\vec{v}$ , expressed in the global coordinate system
<code>vector2dUniformLoadLocal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector2dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector2dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local system
<code>vector2dPointLoadGlobal(p,v)</code>	applies a punctual force to the element; 2D vector $\vec{p}$ defines the global coordinates of the point of application of the force; 2D vector $\vec{v}$ defines the force value and orientation (in global coordinates)
<code>vector2dPointLoadLocal(p,v)</code>	

---

continued on next page ...

Methods ...continued from previous page

	applies a punctual force to the element; 2D vector $\vec{p}$ defines the coordinates of the point of application of the force; 2D vector $\vec{v}$ defines the force value and orientation; both vectors are expressed in the element local-coordinate system
<code>vector3dUniformLoadGlobal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector $\vec{v}$ , expressed in the global coordinate system
<code>vector3dUniformLoadLocal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector3dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector3dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local-coordinate system
<code>vector3dPointLoadGlobal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the global coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation (in global coordinates)
<code>vector3dPointLoadLocal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation; both vectors are expressed in the element local-coordinate system
<code>strainLoad(PlanoDeformacion1,PlanoDeformacion2)</code>	This function creates an imposed strain load in the current load case. The first argument defines the deformation at element start and the second at the element's end.
<code>getCooPuntos(ndiv)</code>	returns $ndiv - 1$ equally-spaced points on the element
<code>getDim()</code>	returns element dimension
<code>getMaterial()</code>	returns the material associated with the element
<code>getL</code>	returns the element length
<code>area</code>	cross-sectional area of the element

continued on next page ...

Methods ...continued from previous page

---

<code>getN</code>	Returns the internal axial force $N$ in the element
-------------------	---

---

#### 4.2.2 TrussSection

This class is used to constructs a truss element object defined by two nodes connected by means of a previously defined section.

```
preprocessor=xc.ProblemaEF().getModelador
trussSectionElement=preprocessor.getElementLoader.newElement(
"truss_section",xc.ID([Nd1Tag,Nd2Tag]))
```

<code>Nd1Tag,Nd2Tag</code>	tags of the nodes connected by the element
----------------------------	--

#### Parameters

---

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getCoordTransf()</code>	returns the identifier of the coordinate-transformation associated with the element

---

#### Methods

---

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not
<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes

---

continued on next page ...

Methods ...continued from previous page

<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation
<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	returns the element centroid coordinates <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions <code>getLongTributaria True</code> to consider the initial geometry shape, <code>False</code> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code>	
<code>calculaLongsTributarias(geomInicial)</code>	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument
<code>getLongTributariaByTag(tag)</code>	returns the tributary length associated with the node labelled with the <code>tag</code> given as argument
<code>calculaAreasTributarias(geomInicial)</code>	returns the tributary area associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getAreaTributaria(Node)</code>	returns the tributary area associated with the <code>Node</code> given as argument

continued on next page ...

Methods ...continued from previous page

---

<code>getAreaTributariaByTag(tag)</code>	returns the tributary area associated with the node labelled with the <code>tag</code> given as argument
<code>calculaVolsTributarios(geomInicial)</code>	returns the tributary volume associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getVolTributario(Node)</code>	returns the tributary volume associated with the <code>Node</code> given as argument
<code>getVolTributarioByTag(tag)</code>	returns the tributary volume associated with the node labelled with the <code>tag</code> given as argument
<code>getMaxCooNod(axisIdx)</code>	returns the maximum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMinCooNod(axisIdx)</code>	returns the minimum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMEDCellType()</code>	interface with MED data format for Salome
<code>vector2dUniformLoadGlobal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector $\vec{v}$ , expressed in the global coordinate system
<code>vector2dUniformLoadLocal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector2dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector2dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local system
<code>vector2dPointLoadGlobal(p,v)</code>	

---

continued on next page ...



Methods ...continued from previous page

---

	applies a punctual force to the element; 2D vector $\vec{p}$ defines the global coordinates of the point of application of the force; 2D vector $\vec{v}$ defines the force value and orientation (in global coordinates)
<code>vector2dPointLoadLocal(p,v)</code>	applies a punctual force to the element; 2D vector $\vec{p}$ defines the coordinates of the point of application of the force; 2D vector $\vec{v}$ defines the force value and orientation; both vectors are expressed in the element local-coordinate system
<code>vector3dUniformLoadGlobal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector $\vec{v}$ , expressed in the global coordinate system
<code>vector3dUniformLoadLocal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector3dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector3dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local-coordinate system
<code>vector3dPointLoadGlobal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the global coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation (in global coordinates)
<code>vector3dPointLoadLocal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation; both vectors are expressed in the element local-coordinate system
<code>strainLoad(PlanoDeformacion1,PlanoDeformacion2)</code>	This function creates an imposed strain load in the current load case. The first argument defines the deformation at element start and the second at the element's end.

---

continued on next page ...

Methods ...continued from previous page

---

<code>getCooPuntos(ndiv)</code>	returns $ndiv - 1$ equally-spaced points on the element
<code>getDim()</code>	returns element dimension
<code>getMaterial()</code>	returns the material associated with the element
<code>getL</code>	returns the element length

### 4.2.3 CorotTruss

This class is used to constructs a corotational truss element object defined by two nodes connected by means of a previously defined uniaxial material.

When constructed with a UniaxialMaterial object, the corotational truss element considers strain-rate effects, and is thus suitable for use as a damping element.

```
preprocessor=xc.ProblemaEF().getModelador
corotTrussElement=preprocessor.getElementLoader.newElement(
"corot_truss",xc.ID([Nd1Tag,Nd2Tag]))
```

Nd1Tag,Nd2Tag                      tags of the nodes connected by the element

#### Parameters

---

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getCoordTransf()</code>	returns the identifier of the coordinate-transformation associated with the element
<code>area</code>	cross-sectional area of the element

#### Methods

---

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not

---

continued on next page ...

Methods ...continued from previous page

<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes
<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation
<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	returns the element centroid coordinates <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions <code>geomInicial = True</code> to consider the initial geometry shape, <code>False</code> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code> <code>calculaLongsTributarias(geomInicial)</code>	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument
<code>getLongTributariaByTag(tag)</code>	returns the tributary length associated with the node labelled with the <code>tag</code> given as argument
<code>calculaAreasTributarias(geomInicial)</code>	returns the tributary area associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape

continued on next page ...

Methods ...continued from previous page

---

<code>getAreaTributaria(Node)</code>	returns the tributary area associated with the <code>Node</code> given as argument
<code>getAreaTributariaByTag(tag)</code>	returns the tributary area associated with the node labelled with the <code>tag</code> given as argument
<code>calculaVolsTributarios(geomInicial)</code>	returns the tributary volume associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getVolTributario(Node)</code>	returns the tributary volume associated with the <code>Node</code> given as argument
<code>getVolTributarioByTag(tag)</code>	returns the tributary volume associated with the node labelled with the <code>tag</code> given as argument
<code>getMaxCooNod(axisIdx)</code>	returns the maximum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMinCooNod(axisIdx)</code>	returns the minimum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMEDCellType()</code>	interface with MED data format for Salome
<code>vector2dUniformLoadGlobal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector $\vec{v}$ , expressed in the global coordinate system
<code>vector2dUniformLoadLocal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector2dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector2dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local system

---

continued on next page ...

Methods ...continued from previous page

- 
- vector2dPointLoadGlobal(p,v)**  
 applies a punctual force to the element; 2D vector  $\vec{p}$  defines the global coordinates of the point of application of the force; 2D vector  $\vec{v}$  defines the force value and orientation (in global coordinates)
- vector2dPointLoadLocal(p,v)**  
 applies a punctual force to the element; 2D vector  $\vec{p}$  defines the coordinates of the point of application of the force; 2D vector  $\vec{v}$  defines the force value and orientation; both vectors are expressed in the element local-coordinate system
- vector3dUniformLoadGlobal(v)**  
 applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector  $\vec{v}$ , expressed in the global coordinate system
- vector3dUniformLoadLocal(v)**  
 applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector  $\vec{v}$ , expressed in the element local coordinate system
- vector3dPointByRelDistLoadGlobal(d,v)**  
 applies a punctual force to the element; scalar  $d$  specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector  $\vec{v}$  defines the force value and orientation, its coordinates are expressed in the global system
- vector3dPointByRelDistLoadLocal(d,v)**  
 applies a punctual force to the element; scalar  $d$  specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector  $\vec{v}$  defines the force value and orientation, its coordinates are expressed in the element local-coordinate system
- vector3dPointLoadGlobal(p,v)**  
 applies a punctual force to the element; 3D vector  $\vec{p}$  defines the global coordinates of the point of application of the force; 3D vector  $\vec{v}$  defines the force value and orientation (in global coordinates)
- vector3dPointLoadLocal(p,v)**  
 applies a punctual force to the element; 3D vector  $\vec{p}$  defines the coordinates of the point of application of the force; 3D vector  $\vec{v}$  defines the force value and orientation; both vectors are expressed in the element local-coordinate system
- strainLoad(PlanoDeformacion1,PlanoDeformacion2)**  
 This function creates an imposed strain load in the current load case. The first argument defines the deformation at element start and the second at the element's end.
- 

continued on next page ...

Methods ...continued from previous page

---

<code>getCooPuntos(ndiv)</code>	returns $ndiv - 1$ equally-spaced points on the element
<code>getDim()</code>	returns element dimension
<code>getMaterial()</code>	returns the material associated with the element
<code>getN</code>	Returns the internal axial force $N$ in the element

#### 4.2.4 CorotTrussSection

This class is used to constructs a corotational truss element object defined by two nodes connected by means of a previously defined section.

```
preprocessor=xc.ProblemaEF().getModelador
corotTrussSectionElement=preprocessor.getElementLoader.newElement(
"corot_truss_section",xc.ID([Nd1Tag,Nd2Tag]))
```

`Nd1Tag,Nd2Tag`                      tags of the nodes connected by the element

#### Parameters

---

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getCoordTransf()</code>	returns the identifier of the coordinate-transformation associated with the element

#### Methods

---

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not

---

continued on next page ...

Methods ...continued from previous page

<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes
<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation
<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	returns the element centroid coordinates <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions <code>getLongTributaria True</code> to consider the initial geometry shape, <code>False</code> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code> <code>calculaLongsTributarias(geomInicial)</code>	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument
<code>getLongTributariaByTag(tag)</code>	returns the tributary length associated with the node labelled with the <code>tag</code> given as argument
<code>calculaAreasTributarias(geomInicial)</code>	returns the tributary area associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape

continued on next page ...

Methods ...continued from previous page

---

<code>getAreaTributaria(Node)</code>	returns the tributary area associated with the <code>Node</code> given as argument
<code>getAreaTributariaByTag(tag)</code>	returns the tributary area associated with the node labelled with the <code>tag</code> given as argument
<code>calculaVolsTributarios(geomInicial)</code>	returns the tributary volume associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getVolTributario(Node)</code>	returns the tributary volume associated with the <code>Node</code> given as argument
<code>getVolTributarioByTag(tag)</code>	returns the tributary volume associated with the node labelled with the <code>tag</code> given as argument
<code>getMaxCooNod(axisIdx)</code>	returns the maximum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMinCooNod(axisIdx)</code>	returns the minimum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMEDCellType()</code>	interface with MED data format for Salome
<code>vector2dUniformLoadGlobal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector $\vec{v}$ , expressed in the global coordinate system
<code>vector2dUniformLoadLocal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector2dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector2dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local system

---

continued on next page ...



Methods ...continued from previous page

- 
- vector2dPointLoadGlobal(p,v)**  
 applies a punctual force to the element; 2D vector  $\vec{p}$  defines the global coordinates of the point of application of the force; 2D vector  $\vec{v}$  defines the force value and orientation (in global coordinates)
- vector2dPointLoadLocal(p,v)**  
 applies a punctual force to the element; 2D vector  $\vec{p}$  defines the coordinates of the point of application of the force; 2D vector  $\vec{v}$  defines the force value and orientation; both vectors are expressed in the element local-coordinate system
- vector3dUniformLoadGlobal(v)**  
 applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector  $\vec{v}$ , expressed in the global coordinate system
- vector3dUniformLoadLocal(v)**  
 applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector  $\vec{v}$ , expressed in the element local coordinate system
- vector3dPointByRelDistLoadGlobal(d,v)**  
 applies a punctual force to the element; scalar  $d$  specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector  $\vec{v}$  defines the force value and orientation, its coordinates are expressed in the global system
- vector3dPointByRelDistLoadLocal(d,v)**  
 applies a punctual force to the element; scalar  $d$  specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector  $\vec{v}$  defines the force value and orientation, its coordinates are expressed in the element local-coordinate system
- vector3dPointLoadGlobal(p,v)**  
 applies a punctual force to the element; 3D vector  $\vec{p}$  defines the global coordinates of the point of application of the force; 3D vector  $\vec{v}$  defines the force value and orientation (in global coordinates)
- vector3dPointLoadLocal(p,v)**  
 applies a punctual force to the element; 3D vector  $\vec{p}$  defines the coordinates of the point of application of the force; 3D vector  $\vec{v}$  defines the force value and orientation; both vectors are expressed in the element local-coordinate system
- strainLoad(PlanoDeformacion1,PlanoDeformacion2)**  
 This function creates an imposed strain load in the current load case. The first argument defines the deformation at element start and the second at the element's end.
- 

continued on next page ...

Methods ...continued from previous page

---

<code>getCooPuntos(ndiv)</code>	returns $ndiv - 1$ equally-spaced points on the element
<code>getDim()</code>	returns element dimension
<code>getMaterial()</code>	returns the material associated with the element

## 4.3 Beam-column elements

### 4.3.1 ElasticBeam2d

This class is used to constructs a uniaxial element with tension, compression, and bending capabilities. The element has three degrees of freedom at each node: translations in the nodal x and y directions and rotation about the nodal z-axis.

The element is defined by two 2D nodes, a previously-defined coordinate-transformation object and a previously-defined 2D elastic section. The initial strain in the element (`initialStrain`) is given by  $\Delta/L$ , where  $\Delta$  is the difference between the element length,  $L$  (as defined by the 1 and 2 node locations), and the zero strain length.

```
preprocessor=xc.ProblemaEF().getModelador
elasticBeam2dElement=preprocessor.getElementLoader.newElement(
"elastic_beam_2d",xc.ID([Nd1Tag,Nd2Tag]))
```

`Nd1Tag,Nd2Tag`                      tags of the nodes connected by the element

#### Parameters

---

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getCoordTransf()</code>	returns the identifier of the coordinate-transformation associated with the element
<code>sectionProperties</code>	
<code>rho</code>	mass density
<code>h</code>	overall depth
<code>initialStrain</code>	initial strain in the element
<code>getV</code>	returns the shear force at the central section of the element
<code>getV1</code>	returns the shear force at the back end of the element
<code>getV2</code>	returns the shear force at the front end of the element
<code>getN1</code>	returns the axial force at the back end of the element
<code>getN2</code>	returns the axial force at the front end of the element
<code>getM1</code>	returns the bending moment at the back end of the element

---

continued on next page ...

Parameters ...continued from previous page

`getM2` returns the bending moment at the front end of the element

## Methods

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not
<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes
<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation
<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	returns the element centroid coordinates <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions <code>getLongTributaria True</code> to consider the initial geometry shape, <i>False</i> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code>	
<code>calculaLongsTributarias(geomInicial)</code>	

continued on next page ...

Methods ...continued from previous page

---

	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument
<code>getLongTributariaByTag(tag)</code>	returns the tributary length associated with the node labelled with the <code>tag</code> given as argument
<code>calculaAreasTributarias(geomInicial)</code>	returns the tributary area associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getAreaTributaria(Node)</code>	returns the tributary area associated with the <code>Node</code> given as argument
<code>getAreaTributariaByTag(tag)</code>	returns the tributary area associated with the node labelled with the <code>tag</code> given as argument
<code>calculaVolsTributarios(geomInicial)</code>	returns the tributary volume associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getVolTributario(Node)</code>	returns the tributary volume associated with the <code>Node</code> given as argument
<code>getVolTributarioByTag(tag)</code>	returns the tributary volume associated with the node labelled with the <code>tag</code> given as argument
<code>getMaxCooNod(axisIdx)</code>	returns the maximum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMinCooNod(axisIdx)</code>	returns the minimum value among the coordinates in the <code>axisIdx</code> axis of the external nodes associated with the element ( <code>axisIdx</code> adopts the values 0,1,...)
<code>getMEDCellType()</code>	interface with MED data format for Salome
<code>vector2dUniformLoadGlobal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector $\vec{v}$ , expressed in the global coordinate system
<code>vector2dUniformLoadLocal(v)</code>	

---

continued on next page ...

Methods ...continued from previous page

---

	applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector2dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector2dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local system
<code>vector2dPointLoadGlobal(p,v)</code>	applies a punctual force to the element; 2D vector $\vec{p}$ defines the global coordinates of the point of application of the force; 2D vector $\vec{v}$ defines the force value and orientation (in global coordinates)
<code>vector2dPointLoadLocal(p,v)</code>	applies a punctual force to the element; 2D vector $\vec{p}$ defines the coordinates of the point of application of the force; 2D vector $\vec{v}$ defines the force value and orientation; both vectors are expressed in the element local-coordinate system
<code>vector3dUniformLoadGlobal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector $\vec{v}$ , expressed in the global coordinate system
<code>vector3dUniformLoadLocal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector3dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector3dPointByRelDistLoadLocal(d,v)</code>	

---

continued on next page ...

Methods ...continued from previous page

	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local-coordinate system
<code>vector3dPointLoadGlobal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the global coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation (in global coordinates)
<code>vector3dPointLoadLocal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation; both vectors are expressed in the element local-coordinate system
<code>strainLoad(PlanoDeformacion1,PlanoDeformacion2)</code>	This function creates an imposed strain load in the current load case. The first argument defines the deformation at element start and the second at the element's end.
<code>getCooPuntos(ndiv)</code>	returns $ndiv - 1$ equally-spaced points on the element

### 4.3.2 ElasticBeam3d

This class is used to constructs a uniaxial element with tension, compression, torsion, and bending capabilities. The element has six degrees of freedom at each node: translations in the nodal x, y, and z directions and rotations about the nodal x, y, and z axes.

The element is defined by two 3D nodes, a previously-defined coordinate-transformation object and a previously-defined 3D elastic section. The element x-axis is oriented from node I toward node J. The initial strain in the element (`initialStrain`) is given by  $\Delta/L$ , where  $\Delta$  is the difference between the element length,  $L$  (as defined by the 1 and 2 node locations), and the zero strain length.

```
preprocessor=xc.ProblemaEF().getModelador
elasticBeam2dElement=preprocessor.getElementLoader.newElement(
"elastic_beam_3d",xc.ID([Nd1Tag,Nd2Tag]))
```

`Nd1Tag,Nd2Tag`

tags of the nodes connected by the element

## Parameters

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getCoordTransf()</code>	returns the identifier of the coordinate-transformation associated with the element
<b>sectionProperties</b>	
<code>rho</code>	mass density
<code>initialStrain</code>	initial strain in the element
<code>getAN2</code>	returns the axial force applied to the front end of the element
<code>getN1</code>	returns the axial force at the back end of the element
<code>getN2</code>	returns the axial force at the front end of the element
<code>getN</code>	returns the mean value of the axial force at the the element $N = \frac{N_1 + N_2}{2}$
<code>getAMz1</code>	returns the bending moment about the local Z axis applied to the back end of the element
<code>getAMz2</code>	returns the bending moment about the local Z axis applied to the front end of the element
<code>getMz1</code>	returns the bending moment about the local Z axis at the back end of the element
<code>getMz2</code>	returns the bending moment about the local Z axis at the front end of the element
<code>getMy1</code>	returns the bending moment about the local Y axis at the back end of the element
<code>getMy2</code>	returns the bending moment about the local Y axis at the front end of the element
<code>getVy</code>	returns the element mean shear force in the local Y axis direction
<code>getVy1</code>	returns the shear force in local Y axis direction at the back end of the element
<code>getVy2</code>	returns the shear force in local Y axis direction at the front end of the element
<code>getAVy1</code>	returns the shear force in local Y axis appliend on the back end of the element
<code>getAVy2</code>	returns the shear force in local Y axis direction applied on the front end of the element
<code>getVz</code>	returns the element mean shear force in the local Z axis direction
<code>getVz1</code>	returns the shear force in local Z axis direction at the back end of the element
<code>getVz2</code>	returns the shear force in local Z axis direction at the front end of the element
<code>getAVz1</code>	returns the shear force in local Z axis appliend on the back end of the element
<code>getAVz2</code>	returns the shear force in local Z axis direction applied on the front end of the element

## Methods

---

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not
<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes
<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation
<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	returns the element centroid coordinates <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions <code>getLongTributaria True</code> to consider the initial geometry shape, <code>False</code> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code>	
<code>calculaLongsTributarias(geomInicial)</code>	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument

---

continued on next page ...



Methods ...continued from previous page

---

**getLongTributariaByTag(tag)**  
returns the tributary length associated with the node labelled with the **tag** given as argument

**calculaAreasTributarias(geomInicial)**  
returns the tributary area associated with each node of the element; the parameter **geomInicial = True** to consider the initial geometry shape in the calculation or **geomInicial = False** for the deformed geometry shape

**getAreaTributaria(Node)**  
returns the tributary area associated with the **Node** given as argument

**getAreaTributariaByTag(tag)**  
returns the tributary area associated with the node labelled with the **tag** given as argument

**calculaVolsTributarios(geomInicial)**  
returns the tributary volume associated with each node of the element; the parameter **geomInicial = True** to consider the initial geometry shape in the calculation or **geomInicial = False** for the deformed geometry shape

**getVolTributario(Node)**  
returns the tributary volume associated with the **Node** given as argument

**getVolTributarioByTag(tag)**  
returns the tributary volume associated with the node labelled with the **tag** given as argument

**getMaxCooNod(axisIdx)** returns the maximum value among the coordinates in the **axisIdx** axis of the external nodes associated with the element (**axisIdx** adopts the values 0,1,...)

**getMinCooNod(axisIdx)** returns the minimum value among the coordinates in the **axisIdx** axis of the external nodes associated with the element (**axisIdx** adopts the values 0,1,...)

**getMEDCellType()** interface with MED data format for Salome

**vector2dUniformLoadGlobal(v)**  
applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector  $\vec{v}$ , expressed in the global coordinate system

**vector2dUniformLoadLocal(v)**  
applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector  $\vec{v}$ , expressed in the element local coordinate system

**vector2dPointByRelDistLoadGlobal(d,v)**

---

continued on next page ...

Methods ...continued from previous page

---

	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector2dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local system
<code>vector2dPointLoadGlobal(p,v)</code>	applies a punctual force to the element; 2D vector $\vec{p}$ defines the global coordinates of the point of application of the force; 2D vector $\vec{v}$ defines the force value and orientation (in global coordinates)
<code>vector2dPointLoadLocal(p,v)</code>	applies a punctual force to the element; 2D vector $\vec{p}$ defines the coordinates of the point of application of the force; 2D vector $\vec{v}$ defines the force value and orientation; both vectors are expressed in the element local-coordinate system
<code>vector3dUniformLoadGlobal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector $\vec{v}$ , expressed in the global coordinate system
<code>vector3dUniformLoadLocal(v)</code>	applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector3dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector3dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local-coordinate system

---

continued on next page ...

Methods ...continued from previous page

---

<code>vector3dPointLoadGlobal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the global coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation (in global coordinates)
<code>vector3dPointLoadLocal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation; both vectors are expressed in the element local-coordinate system
<code>strainLoad(PlanoDeformacion1,PlanoDeformacion2)</code>	This function creates an imposed strain load in the current load case. The first argument defines the deformation at element start and the second at the element's end.
<code>getCooPuntos(ndiv)</code>	returns $ndiv - 1$ equally-spaced points on the element
<code>getVDirEjeFuerteLocales()</code>	returns a vector, expressed in local coordinates, that defines the strong axis orientation
<code>getVDirEjeDebilLocales()</code>	returns a vector, expressed in local coordinates, that defines the weak axis orientation
<code>getAnguloEjeFuerte()</code>	returns the angle between the strong axis and the plane XZ of the element
<code>getAnguloEjeDebil()</code>	returns the angle between the weak axis and the plane XZ of the element
<code>getVDirEjeFuerteGlobales()</code>	returns a vector, expressed in global coordinates, that defines the strong axis orientation
<code>getVDirEjeDebilGlobales()</code>	returns a vector, expressed in global coordinates, that defines the weak axis orientation

### 4.3.3 ForceBeamColumn2d

This command is used to constructs a 2D forceBeamColumn element object, which is based on the iterative force-based formulation. A variety of numerical integration options can be used in the element state determination and encompass both distributed plasticity and plastic hinge integration. More details on the available numerical integration options can be found in the paper titled “Numerical Integration Options for the Force-Based Beam-Column Element in OpenSees”, by Michael H. Scott.

The element is defined by two 2D nodes, a previously-defined coordinate-transformation object and a previously-defined 2D elastic section.

```
preprocessor=xc.ProblemaEF().getModelador
elasticBeam2dElement=preprocessor.getElementLoader.newElement(
```

```
"force_beam_column_2d",xc.ID([Nd1Tag,Nd2Tag]))
```

Nd1Tag,Nd2Tag            tags of the nodes connected by the element

## Parameters

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getCoordTransf()</code>	returns the identifier of the coordinate transformation associated with the element
<code>rho</code>	mass density

## Methods

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not
<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes
<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation
<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	returns the element centroid coordinates

continued on next page ...

Methods ...continued from previous page

---

	<code>geomInicial = True</code> to consider the initial geometry shape
	<code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions
	<code>getLongTributaria</code> <i>True</i> to consider the initial geometry shape, <i>False</i> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code>	
<code>calculaLongsTributarias(geomInicial)</code>	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument
<code>getLongTributariaByTag(tag)</code>	returns the tributary length associated with the node labelled with the <code>tag</code> given as argument
<code>calculaAreasTributarias(geomInicial)</code>	returns the tributary area associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getAreaTributaria(Node)</code>	returns the tributary area associated with the <code>Node</code> given as argument
<code>getAreaTributariaByTag(tag)</code>	returns the tributary area associated with the node labelled with the <code>tag</code> given as argument
<code>calculaVolsTributarios(geomInicial)</code>	returns the tributary volume associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getVolTributario(Node)</code>	returns the tributary volume associated with the <code>Node</code> given as argument
<code>getVolTributarioByTag(tag)</code>	returns the tributary volume associated with the node labelled with the <code>tag</code> given as argument

---

continued on next page ...

Methods ...continued from previous page

---

`getMaxCooNod(axisIdx)` returns the maximum value among the coordinates in the `axisIdx` axis of the external nodes associated with the element (`axisIdx` adopts the values 0,1,...)

`getMinCooNod(axisIdx)` returns the minimum value among the coordinates in the `axisIdx` axis of the external nodes associated with the element (`axisIdx` adopts the values 0,1,...)

`getMEDCellType()` interface with MED data format for Salome

`vector2dUniformLoadGlobal(v)`  
applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector  $\vec{v}$ , expressed in the global coordinate system

`vector2dUniformLoadLocal(v)`  
applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector  $\vec{v}$ , expressed in the element local coordinate system

`vector2dPointByRelDistLoadGlobal(d,v)`  
applies a punctual force to the element; scalar  $d$  specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector  $\vec{v}$  defines the force value and orientation, its coordinates are expressed in the global system

`vector2dPointByRelDistLoadLocal(d,v)`  
applies a punctual force to the element; scalar  $d$  specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector  $\vec{v}$  defines the force value and orientation, its coordinates are expressed in the element local system

`vector2dPointLoadGlobal(p,v)`  
applies a punctual force to the element; 2D vector  $\vec{p}$  defines the global coordinates of the point of application of the force; 2D vector  $\vec{v}$  defines the force value and orientation (in global coordinates)

`vector2dPointLoadLocal(p,v)`  
applies a punctual force to the element; 2D vector  $\vec{p}$  defines the coordinates of the point of application of the force; 2D vector  $\vec{v}$  defines the force value and orientation; both vectors are expressed in the element local-coordinate system

`vector3dUniformLoadGlobal(v)`  
applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector  $\vec{v}$ , expressed in the global coordinate system

`vector3dUniformLoadLocal(v)`

---

continued on next page ...

Methods ...continued from previous page

	applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector3dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector3dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local-coordinate system
<code>vector3dPointLoadGlobal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the global coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation (in global coordinates)
<code>vector3dPointLoadLocal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation; both vectors are expressed in the element local-coordinate system
<code>strainLoad(PlanoDeformacion1,PlanoDeformacion2)</code>	This function creates an imposed strain load in the current load case. The first argument defines the deformation at element start and the second at the element's end.
<code>getCooPuntos(ndiv)</code>	returns $ndiv - 1$ equally-spaced points on the element
<code>getNumSections</code>	
<code>getSections</code>	Returns element sections

#### 4.3.4 ForceBeamColumn3d

This command is used to constructs a 3D forceBeamColumn element object, which is based on the iterative force-based formulation. A variety of numerical integration options can be used in the element state determination and encompass both distributed plasticity and plastic hinge integration. More details on the available numerical integration options can be found in the paper titled ■Numerical Integration Options for the Force-Based Beam-Column Element in OpenSees■, by Michael H. Scott.

The element is defined by two 3D nodes, a previously-defined coordinate-transformationt object and a previously-defined 3D elastic section.

```

preprocessor=xc.ProblemaEF().getModelador
elasticBeam2dElement=preprocessor.getElementLoader.newElement(
"force_beam_column_3d",xc.ID([Nd1Tag,Nd2Tag]))

```

Nd1Tag,Nd2Tag            tags of the nodes connected by the element

## Parameters

<code>getIdxNodes</code>	vector containing the node index to be used in Vtk graphics
<code>getDimension</code>	element dimension
<code>getVtkCellType</code>	cell type for Vtk graphics
<code>getCoordTransf()</code>	returns the identifier of the coordinate-transformation associated with the element
<code>rho</code>	mass density

## Methods

<code>commitState()</code>	the element is to commit its current state; returns 0 if successful, a negative number if not
<code>revertToLastCommit()</code>	the element is to set its current state to the last committed state; returns 0 if successful, a negative number if not
<code>revertToStart</code>	the element is to set its current state to the state it was at before the analysis started; returns 0 if successful, a negative number if not
<code>getNumDOF</code>	returns the number of DOF associated with the element; this should equal the sum of the DOFs at each of the external nodes
<code>getResistingForce()</code>	returns the resisting force vector for the element; this is equal to the applied load due to element loads minus the loads at the nodes due to internal stresses in the element due to the current trial displacement, i.e. $R_e = P_e - f_{R_e}(U_{trial})$
<code>setDeadSRF</code>	assigns Stress Reduction Factor for element deactivation
<code>getVtkCellType()</code>	returns cell type for Vtk graphics
<code>getMEDCellType()</code>	returns cell type for MED file writing.
<code>getPosCentroid(geomInicial)</code>	returns the element centroid position. <code>geomInicial = True</code> to consider the initial geometry shape <code>geomInicial = False</code> to consider the deformed geometry shape
<code>getCooCentroid(geomInicial)</code>	

continued on next page ...



Methods ...continued from previous page

---

	returns the element centroid coordinates
	<code>geomInicial = True</code> to consider the initial geometry shape
	<code>geomInicial = False</code> to consider the deformed geometry shape
<code>getPoints(ni,nj,nk,geomInicial)</code>	returns a uniform grid of points over the element. <code>ni,nj,nk</code> number of divisions in i,j,k directions
	<code>getLongTributaria</code> <i>True</i> to consider the initial geometry shape, <i>False</i> for the deformed geometry shape
<code>resetTributarias()</code>	reset the tributary length, area and volume of connected nodes
<code>vuelcaTributarias</code>	
<code>calculaLongsTributarias(geomInicial)</code>	returns the tributary length associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getLongTributaria(Node)</code>	returns the tributary length associated with the <code>Node</code> given as argument
<code>getLongTributariaByTag(tag)</code>	returns the tributary length associated with the node labelled with the <code>tag</code> given as argument
<code>calculaAreasTributarias(geomInicial)</code>	returns the tributary area associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getAreaTributaria(Node)</code>	returns the tributary area associated with the <code>Node</code> given as argument
<code>getAreaTributariaByTag(tag)</code>	returns the tributary area associated with the node labelled with the <code>tag</code> given as argument
<code>calculaVolsTributarios(geomInicial)</code>	returns the tributary volume associated with each node of the element; the parameter <code>geomInicial = True</code> to consider the initial geometry shape in the calculation or <code>geomInicial = False</code> for the deformed geometry shape
<code>getVolTributario(Node)</code>	returns the tributary volume associated with the <code>Node</code> given as argument
<code>getVolTributarioByTag(tag)</code>	returns the tributary volume associated with the node labelled with the <code>tag</code> given as argument

---

continued on next page ...

Methods ...continued from previous page

---

`getMaxCooNod(axisIdx)` returns the maximum value among the coordinates in the `axisIdx` axis of the external nodes associated with the element (`axisIdx` adopts the values 0,1,...)

`getMinCooNod(axisIdx)` returns the minimum value among the coordinates in the `axisIdx` axis of the external nodes associated with the element (`axisIdx` adopts the values 0,1,...)

`getMEDCellType()` interface with MED data format for Salome

`vector2dUniformLoadGlobal(v)`  
applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector  $\vec{v}$ , expressed in the global coordinate system

`vector2dUniformLoadLocal(v)`  
applies a uniform surface load to the element; the value and direction of the load is defined by the 2D vector  $\vec{v}$ , expressed in the element local coordinate system

`vector2dPointByRelDistLoadGlobal(d,v)`  
applies a punctual force to the element; scalar  $d$  specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector  $\vec{v}$  defines the force value and orientation, its coordinates are expressed in the global system

`vector2dPointByRelDistLoadLocal(d,v)`  
applies a punctual force to the element; scalar  $d$  specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 2D vector  $\vec{v}$  defines the force value and orientation, its coordinates are expressed in the element local system

`vector2dPointLoadGlobal(p,v)`  
applies a punctual force to the element; 2D vector  $\vec{p}$  defines the global coordinates of the point of application of the force; 2D vector  $\vec{v}$  defines the force value and orientation (in global coordinates)

`vector2dPointLoadLocal(p,v)`  
applies a punctual force to the element; 2D vector  $\vec{p}$  defines the coordinates of the point of application of the force; 2D vector  $\vec{v}$  defines the force value and orientation; both vectors are expressed in the element local-coordinate system

`vector3dUniformLoadGlobal(v)`  
applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector  $\vec{v}$ , expressed in the global coordinate system

`vector3dUniformLoadLocal(v)`

---

continued on next page ...

Methods ...continued from previous page

---

	applies a uniform surface load to the element; the value and direction of the load is defined by the 3D vector $\vec{v}$ , expressed in the element local coordinate system
<code>vector3dPointByRelDistLoadGlobal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the global system
<code>vector3dPointByRelDistLoadLocal(d,v)</code>	applies a punctual force to the element; scalar $d$ specifies the offset distance from node 2 (toward node 1) where the force is applied, this distance is input as a length fraction (its value varies between 0 and 1); 3D vector $\vec{v}$ defines the force value and orientation, its coordinates are expressed in the element local-coordinate system
<code>vector3dPointLoadGlobal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the global coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation (in global coordinates)
<code>vector3dPointLoadLocal(p,v)</code>	applies a punctual force to the element; 3D vector $\vec{p}$ defines the coordinates of the point of application of the force; 3D vector $\vec{v}$ defines the force value and orientation; both vectors are expressed in the element local-coordinate system
<code>strainLoad(PlanoDeformacion1,PlanoDeformacion2)</code>	This function creates an imposed strain load in the current load case. The first argument defines the deformation at element start and the second at the element's end.
<code>getCooPuntos(ndiv)</code>	returns $ndiv - 1$ equally-spaced points on the element
<code>getNumSections</code>	Returns element sections
<code>getVDirEjeFuerteLocales()</code>	returns a vector, expressed in local coordinates, that defines the strong axis orientation
<code>getVDirEjeDebilLocales()</code>	returns a vector, expressed in local coordinates, that defines the weak axis orientation
<code>getAnguloEjeFuerte()</code>	returns the angle between the strong axis and the plane XZ of the element
<code>getAnguloEjeDebil()</code>	returns the angle between the weak axis and the plane XZ of the element

---

continued on next page ...

Methods ...continued from previous page

---

`getVDirEjeFuerteGlobales()`

returns a vector, expressed in global coordinates, that defines the strong axis orientation

`getVDirEjeDebilGlobales()`

returns a vector, expressed in global coordinates, that defines the weak axis orientation

#### **4.3.5 Numerical integration options for the forceBeamColumn elements.**

The following paragraph are based on the Michael H. Scott article in OpenSees.

To specify the numerical integration options to represent distributed plasticity or non-prismatic section details in force-based beam-column elements, i.e., across the entire element domain  $[0, L]$  we can use one of the following schemes.

## Chapter 5

# Loads

### 5.1 Time series

TimeSeries objects represents the relationship between the time in the domain,  $t$ , and the load factor applied to the loads,  $\lambda$ , in the load pattern with which the TimeSeries object is associated, i.e.

$$\lambda = F(t) \tag{5.1}$$

#### 5.1.1 Constant TimeSeries

#### 5.1.2 Linear TimeSeries

#### 5.1.3 Trigonometric TimeSeries

#### 5.1.4 Triangular TimeSeries

#### 5.1.5 Rectangular TimeSeries

#### 5.1.6 Pulse TimeSeries

#### 5.1.7 Path TimeSeries

#### 5.1.8 PeerMotion

#### 5.1.9 PeerNGAMotion

DRAFT

# Chapter 6

## Solution

### 6.1 Analysis and its components

In XC (just like in OpenSees) an analysis is an object which is composed by the aggregation of component objects. It is the component objects which define the type of analysis that is performed on the model. The component classes consist of the following:

Constraint Handler : determines how the constraint equations are enforced in the analysis :  
how it handles the boundary conditions/imposed displacements

DOF\_Numberer: determines the mapping between equation numbers and degrees-of-freedom

Integrator: determines the predictive step for time  $t+dt$

SolutionAlgorithm : determines the sequence of steps taken to solve the non-linear equation at the current time step

SystemOfEqn/Solver: within the solution algorithm, it specifies how to store and solve the system of equations in the analysis

Convergence Test: determines when convergence has been achieved.

#### 6.1.1 Constraint handlers

The ConstraintHandler objects determine how the constraint equations are enforced in the analysis. Constraint equations enforce a specified value for a DOF, or a relationship between DOFs.

##### 6.1.1.1 Constraint types

The types of constraints supported by the software are defined in 2.3.2.3 and 2.3.2.4.

##### 6.1.1.2 Constraint handler types

The available constraint handler types are the following:

- Plain Constraints
- Lagrange Multipliers

- Penalty Method
- Transformation Method

#### 6.1.1.3 Plain Constraints

A plain constraint handler can only enforce homogeneous single point constraints (fix command) and multi-point constraints constructed where the constraint matrix is equal to the identity (equalDOF command).

#### 6.1.1.4 Lagrange multipliers

A Lagrange multiplier constraint handler enforces the constraints by introducing Lagrange multipliers to the system of equations.

In an object of this type the following parameters can be defined:

$\alpha_S$ : factor on single point, optional, default = 1.0

$\alpha_M$ : factor on multi-points, optional default = 1.0

The Lagrange multiplier method introduces new unknowns to the system of equations. The diagonal part of the system corresponding to these new unknowns is 0.0. This ensure that the system IS NOT symmetric positive definite.

**6.1.1.4.1 LagrangeMP\_FE** LagrangeMP\_FE is a subclass of FE\_Element used to enforce a multi point constraint, of the form  $U_c = C_{cr} U_r$ , where  $U_c$  are the constrained degrees-of-freedom at the constrained node,  $U_r$  are the retained degrees-of-freedom at the retained node and  $C_{cr}$  a matrix defining the relationship between these degrees-of-freedom.

To enforce the constraint the following are added to the tangent and the residual:

$$\begin{bmatrix} 0 & \alpha C^t \\ \alpha C & 0 \end{bmatrix}, \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

at the locations corresponding to the constrained degree-of-freedom specified by the MP\_Constraint, i.e.  $[U_c \ U_r]$ , and the lagrange multiplier degrees-of-freedom introduced by the LagrangeConstraintHandler for this constraint,  $C = [-I \ C_{cr}]$ . Nothing is added to the residual.

To constructs a LagrangeMP\_FE element to enforce the constraint specified by the MP\_Constraint *theMP* using a default value for  $\alpha$  of *alpha*. The FE\_Element class constructor is called with the integers 3 and the two times the size of the *retainedID* plus the size of the *constrainedID* at the MP\_Constraint *theMP* plus . A Matrix and a Vector object are created for adding the contributions to the tangent and the residual. The residual is zeroed. If the MP\_Constraint is not time varying, then the contribution to the tangent is determined. Links are set to the retained and constrained nodes. The DOF\_Group tag ID is set using the tag of the constrained Nodes DOF\_Group, the tag of the retained Node Dof\_group and the tag of the LagrangeDOF\_Group, *theGroup*. A warning message is printed and the program is terminated if either not enough memory is available for the Matrices and Vector or the constrained and retained Nodes of their DOF\_Groups do not exist.

*virtual void setID(void);*

Causes the LagrangeMP\_FE to determine the mapping between it's equation numbers and the



degrees-of-freedom. This information is obtained by using the mapping information at the DOF\_Group objects associated with the constrained and retained nodes and the LagrangeDOF\_Group, *theGroup*. Returns 0 if successful. Prints a warning message and returns a negative number if an error occurs: -2 if the Node has no associated DOF\_Group, -3 if the constrained DOF specified is invalid for this Node (sets corresponding ID component to -1 so nothing is added to the tangent) and -4 if the ID in the DOF\_Group is too small for the Node (again setting corresponding ID component to -1).

*virtual const Matrix &getTangent(Integrator \*theIntegrator);*

If the MP\_Constraint is time-varying, from the MP\_Constraint *theMP* it obtains the current  $C_{cr}$  matrix; it then adds the contribution to the tangent matrix. Returns this tangent Matrix.

*virtual const Vector &getResidual(Integrator \*theIntegrator);*

Returns the residual, a zero Vector.

#### 6.1.1.5 Penalty method

A penalty constraint handler enforces the constraints using the penalty method. These handlers allows the user to choose the penalty factors:

$\alpha_S$ : penalty factor on single point constraints

$\alpha_M$ : penalty factor on multiple point constraints

The degree to which the constraints are enforced is dependent on the penalty values chosen. Problems can arise if these values are too small (constraint not enforced strongly enough) or too large (problems associated with conditioning of the system of equations).

#### 6.1.1.6 Transformation method

A transformation constraint handler enforces the constraints using the transformation method. The single-point constraints when using the transformation method are done directly. The matrix equation is not manipulated to enforce them, rather the trial displacements are set directly at the nodes at the start of each analysis step.

Great care must be taken when multiple constraints are being enforced as the transformation method does not follow constraints:

1. If a node is fixed, constrain it with the fix command and not equalDOF or other type of constraint.
2. If multiple nodes are constrained, make sure that the retained node is not constrained in any other constraint.

And remember if a node is constrained to multiple nodes in your model it probably means you have messed up.

### 6.1.2 DOF\_Numberer: mapping between equation numbers and degrees of freedom

The DOF\_Numberer object determines the mapping between equation numbers and degrees of freedom i.e. how degrees of freedom are numbered.

#### **6.1.2.1 Plain Numberers**

WIP Work in progress...

#### **6.1.2.2 Reverse Cuthill-McKee Numberers**

WIP Work in progress...

### **6.1.3 System of equation and its solution**

This objects are used to construct the LinearSOE and LinearSolver objects to store and solve the system of equations in the analysis.

#### **6.1.3.1 Band general system of equations**

BandGeneralSOE WIP Work in progress...

#### **6.1.3.2 Band symmetric positive definite system of equations**

BandSPDSOE WIP Work in progress...

#### **6.1.3.3 Profile symmetric positive definite system of equations**

ProfileSPDSOE WIP Work in progress...

#### **6.1.3.4 Sparse general linear system of equations (SuperLU)**

SuperLUSOE WIP Work in progress...

#### **6.1.3.5 Sparse general linear system of equations (UmfPack)**

UmfPackSOE WIP Work in progress...

#### **6.1.3.6 Full general linear system of equations**

UmfPackSOE WIP Work in progress...

#### **6.1.3.7 Sparse symmetric system of equations**

SparseSYM WIP Work in progress...

### **6.1.4 Integrator**

The Integrator object determines the meaning of the terms in the system of equation object  $Ax=B$ . The Integrator object is used for the following:

- determine the predictive step for time  $t+dt$
- specify the tangent matrix and residual vector at any iteration
- determine the corrective step based on the displacement increment  $dU$

#### 6.1.4.1 Static integrators

6.1.4.1.1 Load Control WIP Work in progress...

6.1.4.1.2 Displacement Control WIP Work in progress...

6.1.4.1.3 Minimum Unbalanced Displacement Norm WIP Work in progress...

6.1.4.1.4 Arc-Length Control

#### 6.1.4.2 Transient integrators

6.1.4.2.1 Central Difference WIP Work in progress...

6.1.4.2.2 Newmark Method WIP Work in progress...

6.1.4.2.3 Hilber-Hughes-Taylor Method WIP Work in progress...

6.1.4.2.4 Generalized Alpha Method WIP Work in progress...

6.1.4.2.5 TRBDF2

### 6.1.5 Convergence test

The convergence tests are used to allow certain SolutionAlgorithm objects to determine if convergence has been achieved at the end of an iteration step. The convergence test is applied to the matrix equation,  $AX=B$  stored in the LinearSOE.

#### 6.1.5.1 Norm Unbalance Test

WIP Work in progress...

#### 6.1.5.2 Norm Displacement Increment Test

WIP Work in progress...

#### 6.1.5.3 Energy Increment Test

WIP Work in progress...

#### 6.1.5.4 Relative Norm Unbalance Test

WIP Work in progress...

#### 6.1.5.5 Relative Norm Displacement Increment Test

WIP Work in progress...

#### 6.1.5.6 Total Relative Norm Displacement Increment Test

WIP Work in progress...

#### **6.1.5.7 Relative Energy Increment Test**

WIP Work in progress...

#### **6.1.5.8 Fixed Number of Iterations**

WIP Work in progress...

### **6.1.6 Solution algorithm**

A SolutionAlgorithm object is used to determine the sequence of steps taken to solve the non-linear equation.

#### **6.1.6.1 Linear Algorithm**

WIP Work in progress...

#### **6.1.6.2 Newton Algorithm**

WIP Work in progress...

#### **6.1.6.3 Newton with Line Search Algorithm**

WIP Work in progress...

#### **6.1.6.4 Modified Newton Algorithm**

WIP Work in progress...

#### **6.1.6.5 Krylov-Newton Algorithm**

WIP Work in progress...

#### **6.1.6.6 Secant Newton Algorithm**

WIP Work in progress...

#### **6.1.6.7 BFGS Algorithm**

WIP Work in progress...

#### **6.1.6.8 Broyden Algorithm**

WIP Work in progress...

### **6.1.7 Analyze method**

All analysis objects have an 'analyze' method that is used to perform the analysis. This command can receive one or more of the following parameters:

numSteps: number of analysis steps to perform.

dt: time-step increment. Required if transient or variable transient analysis

dtMin,dtMax: minimum and maximum time steps. Required if a variable time step transient analysis was specified.

Jd: number of iterations user would like performed at each step. The variable transient analysis will change current time step if last analysis step took more or less iterations than this to converge. Required if a variable time step transient analysis was specified.

This command returns a zero if successful or a negative value otherwise.

DRAFT

DRAFT

# Chapter 7

## Check routines

### 7.1 Introduction

This chapter describes the routines that can be used to check the design following the specifications of different design codes.

### 7.2 Check routines for steel

#### 7.2.1 Lateral torsional buckling of steel beams (EC3)

Flexural buckling check, according to article 5.5.2 of EC3, can be written as:

$$F = \frac{M_d}{M_{b,Rd}} \leq 1 \quad (7.1)$$

where:

$M_d$  Design value of bending moment.

$M_{b,Rd}$  Buckling resistance

##### 7.2.1.1 Design lateral torsional buckling resistance $M_{b,Rd}$

Design value of lateral torsional buckling resistance can be determined as follows:

$$M_{b,Rd} = \chi_{LT} \cdot W_y \cdot \frac{f_y}{\gamma_{M1}} \quad (7.2)$$

where  $\gamma_{M1}$  is the partial factor for member buckling resistance and  $f_y$  is the characteristic value of the yield strength.

##### 7.2.1.1.1 Cross section modulus $W_y$

$$W_y = \begin{cases} W_{pl,y} & : 1 \text{ or } 2 \text{ class cross section (plastic section modulus)} \\ W_{el,y} & : \text{class 3 cross section (elastic section modulus)} \\ W_{eff,y} & : \text{class 4 cross section (effective section modulus)} \end{cases} \quad (7.3)$$

**7.2.1.1.2 Reduction factor  $\chi_{LT}$**  The reduction factor  $\chi_{LT}$  for the appropriate non-dimensional slenderness  $\lambda_{LT}$  may be determined from:

$$\chi_{LT} = \frac{1}{\phi_{LT} + \sqrt{\phi_{LT}^2 - \bar{\lambda}_{LT}^2}} \quad (7.4)$$

**Intermediate factor  $\phi_{LT}$**  calculated as:

$$\phi_{LT} = 0.5[1 + \alpha_{LT} \cdot (\bar{\lambda}_{LT} - 0.2) + \bar{\lambda}_{LT}^2] \quad (7.5)$$

**Imperfection factor  $\alpha_{LT}$**  Depending on the buckling curves <sup>1</sup>:

$$\alpha_{LT} = \begin{cases} 0.21 & : \text{curve a} \\ 0.34 & : \text{curve b} \\ 0.49 & : \text{curve c} \\ 0.76 & : \text{curve d} \end{cases} \quad (7.6)$$

**Non-dimensional beam slenderness  $\bar{\lambda}_{LT}$**  calculated as:

$$\bar{\lambda}_{LT} = \sqrt{\frac{W_y f_y}{M_{cr}}} \quad (7.7)$$

Where the non-dimensional slenderness  $\bar{\lambda}_{LT} \leq 0.4$  no allowance for lateral-torsional buckling is necessary<sup>2</sup>

**Critical bending moment  $M_{cr}$**  Following the [2] reference, the elastic critical moment is directly affected by the following factors:

- Material properties (elastic modulus, poisson ratio,...).
- Properties of the cross section (torsion constant, warping constant<sup>3</sup> and moment of inertia about the minor axis).
- Properties of the beam such as its length, lateral bending and warping restrictions at supports.
- Loading: lateral-torsional buckling is greatly affected by moment diagram and loading position with respect to the section shear centre.

As a generally accepted procedure, consideration of the bending moment diagram is taken into account by means of the *equivalent uniform moment factor (EUMF)*, also called the *moment gradient correction factor*. The elastic critical moment for any bending moment diagram is obtained by multiplying this factor by the elastic critical moment of the simply supported beam with uniform moment.

The expression of the elastic critical moment is calculated as:

$$M_{cr} = C_1 \frac{\pi^2 E I_z}{(k_z L)^2} \sqrt{\left(\frac{k_z}{k_w}\right)^2 \frac{I_w}{I_z} + \frac{(k_z L)^2 G I_t}{\pi^2 E I_z}} \quad (7.8)$$

<sup>1</sup>Selection of flexural buckling curve for a cross section can be made from EC3 tables.

<sup>2</sup>This value can be adapted in the national annex.

<sup>3</sup>The effect of torsional loading can be split into two parts, the first part causing twist and the second, *warping*.



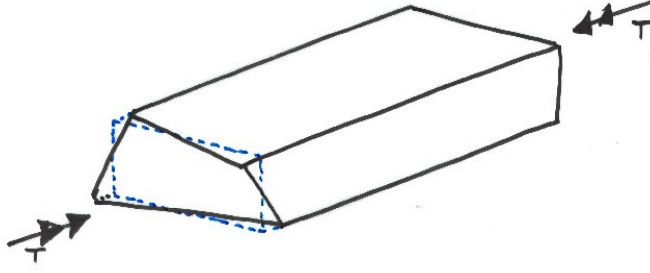


Figure 7.1: Warping on a rectangular section due to pure torsion

where the lateral bending coefficient  $k_z$  and the warping coefficient  $k_w$  are introduced to consider support conditions different from the simply supported beam. These coefficients are equal to 1 for free lateral bending and free warping and 0.5 for prevented lateral bending and prevented warping. The rest of the parameters are defined as follows:

$E$  Elastic modulus.

$G$  Shear modulus.

$k_z L$  Lateral torsional buckling length of the beam: length between points which have lateral restraint.

$I_z$  Moment of inertia about the minor axis.

$I_t$  Torsional constant.

$I_w$  Warping constant.

Finally, the moment moment gradient factor  $C_1$  is calculated as:

$$C_1 = \frac{\sqrt{\sqrt{k}A_1 + \left[\frac{(1-\sqrt{k})}{2}A_2\right]^2 + \frac{(1-\sqrt{k})}{2}A_2}}{A_1} \quad (7.9)$$

where:

$k = \sqrt{k_1 k_2}$  depends on the lateral bending and warping conditions coefficients  $k_1$  and  $k_2$ .

$M_i$  and  $M_{max}$  moments which represent moment diagram of the beam as in figure 7.2.  $M_{max}$  is the maximum moment and  $M_1, M_2, M_3, M_4$  and  $M_5$  are the values of the moment at the different sections of the beam as indicated in the figure, each of them with the corresponding sign.

$$A_1 = \frac{M_{max}^2 + \sum_{i=1..5} \alpha_i M_i^2}{(1 + \sum_{i=1..5} \alpha_i) \cdot M_{max}^2}$$

$$A_2 = \frac{M_1 + 2M_2 + 3M_3 + 2M_4 + M_5}{9 \cdot M_{max}}$$

$\alpha_i$  factor calculated as follows:

$$\alpha_1 = (1 - k_2); \alpha_2 = 5 \frac{k_1^3}{k_2^2}; \alpha_3 = 5 \left( \frac{1}{k_1} + \frac{1}{k_2} \right); \alpha_4 = 5 \frac{k_2^3}{k_1^2}; \alpha_5 = (1 - k_1) \quad (7.10)$$

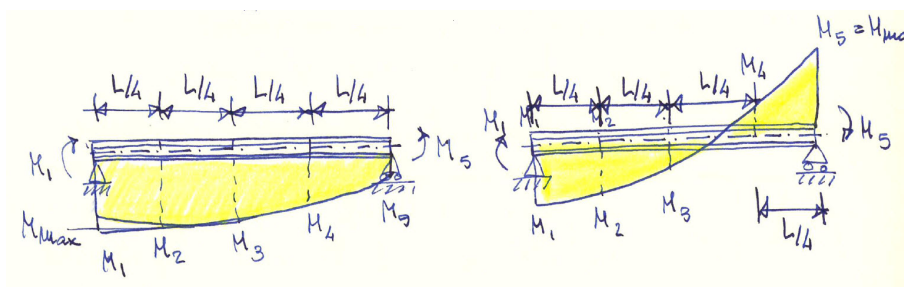


Figure 7.2: Moment diagrams and moment values for calculation of  $C_1$

$k_1$  and  $k_2$  support conditions at each end of the beam as follows:

$k_1 = 1.0$  Free lateral bending and warping at left end.

$k_1 = 0.5$  Prevented support conditions at left end.

$k_2 = 1.0$  Free lateral bending and warping at right end.

$k_2 = 0.5$  Prevented support conditions at right end.

## 7.3 Check routines for reinforced concrete

### 7.3.1 Verification of RC sections

#### 7.3.1.1 Sections definition

#### 7.3.1.2 Limit State at Failure under normal stresses verification

**7.3.1.2.1 lanzaCalculoTNFromXCData** This function carries out the verification of the limit state at failure under normal stresses. It takes as input the internal forces and bending moments calculated for the shell elements for every ULS combinations, the sections definition and the interaction diagrams to be employed.

The function returns two files with the verification results: `outputFileName.py`: XC file that assigns each shell element the capacity factor (worst-case) FCC calculated for reinforcement in directions 1 and 2, together with the concomitant axial force and bending moments  $N$   $M_y$   $M_z$ . `outputFileName.py`:  $\LaTeX$  file containing a table with the following items:

#### Section 1

Element number	Section name	ULS name	Axial force NCP1	Bending moment MyCP1	Bending moment MzCP1	Capacity factor FCCP1
...	...	...	...	...	...	...

#### Section 2

Element number	Section name	ULS name	Axial force NCP2	Bending moment MyCP2	Bending moment MzCP2	Capacity factor FCCP2
...	...	...	...	...	...	...

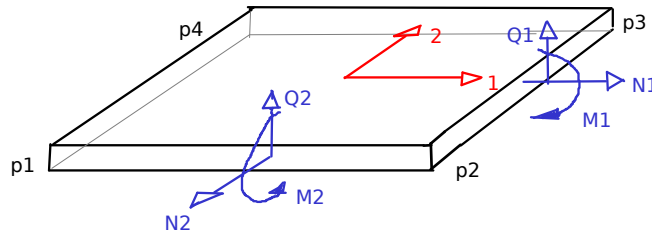


Figure 7.3: Positive directions of forces and moments in shell elements

```
from materials.xLamina import calculo_tn
calculo_tn.lanzaCalculoTNFromXCData(preprocessor,analysis,intForcCombFileName,outputFileName,
mapSectionsForEveryElement,mapSectionsDefinition, mapInteractionDiagrams)
```

preprocessor	preprocessor name
analysis	type of analysis to be performed. Some predefined types can be selected: from solution import predefined_solutions as ps pEF=xc.ProblemaEF() ps.simple_static_linear(pEF) for linear static analysis ps.simple_newton_raphson(pEF) for non linear analysis using Newton Raphson algorithm to solve the equation ps.simple_newton_raphson_band_gen(pEF) ps.simple_static_modified_newton(pEF) for non linear analysis using the modified Newton Raphson algorithm to solve the equation ps.penalty_newton_raphson(pEF) ps.frequency_analysis(p)
intForcCombFileName	name of the file where are to be found the shell element forces and moments obtained for each ULS combination, to be used in the verification of the ULS under normal stresses. Directory path, file name and extension must be specified. The file contains a list of the following items (see the positive directions of the forces and moments in figure 7.3): $ULS_{id} \quad elem_{id} \quad N_1 \quad N_2 \quad N_{12} \quad M_1 M_2 \quad M_{12} \quad Q_1 \quad Q_2$
outputFileName	name of the output files (directory path and file name without extension) where to write the results of the verification
mapSectionsForEveryElement	data structure, such that, for each shell element: <code>mapSectionsForEveryElement[tagElem]=[nmbScc1,nmbScc2]</code> , where <code>nmbScc1</code> and <code>nmbScc2</code> are the names identifying the sections to be used in 1 and 2 directions respectively
mapSectionsDefinition	

data structure that links each section name and direction with the respective record that contains the section definition parameters

`mapInteractionDiagrams`

data structure that links each section name and direction with the interaction diagram object to be used for its verification

### 7.3.1.3 Limit State of Failure due to shear verification

This function carries out the verification of the limit state at failure under normal stresses. It takes as input the internal forces and bending moments calculated for the shell elements for every ULS combinations, the sections definition and the interaction diagrams to be employed.

The function returns two files with the verification results:

`outputFileName.py`: XC file that assigns each shell element the capacity factor (worst-case) FCC calculated for reinforcement in directions 1 and 2, together with the concomitant forces and moments `N My Mz Vy Vz` and the ultimate shear forces and moment `Mu Vcu Vsu Vu` `outputFileName.py`: L<sup>A</sup>T<sub>E</sub>X file containing a table with the following items:

#### Section 1

Element number	Section name	ULS name	Axial force NCP1	Bending moment MyCP1	Bending moment MzCP1	Cracking moment MuCP1	Shear force VyCP1	Shear force VyCP1	Ultimate shear force VuCP1	Capacity factor FCCP1
.....										

#### Section 2

Element number	Section name	ULS name	Axial force NCP2	Bending moment MyCP2	Bending moment MzCP2	Cracking moment MuCP2	Shear force VyCP2	Shear force VyCP2	Ultimate shear force VuCP2	Capacity factor FCCP2
.....										

```
from materials.xLamina import calculo_v
calculo_v.lanzaCalculoV(preprocessor,analysis,intForcCombFileName,outputFileName,
mapSectionsForEveryElement,mapSectionsDefinition,mapInteractionDiagrams,
procesResultVerifV)
```

`preprocessor`  
`analysis`

preprocessor name  
type of analysis to be performed. Some predefined types can be selected:  
from solution import predefined\_solutions as ps  
pEF=xc.ProblemaEF()  
ps.simple\_static\_linear(pEF) for linear static analysis  
ps.simple\_newton\_raphson(pEF) for non linear analysis using Newton Raphson algorithm to solve the equation  
ps.simple\_newton\_raphson\_band\_gen(pEF)

	<code>ps.simple_static_modified_newton(pEF)</code> for non linear analysis using the modified Newton Raphson algorithm to solve the equation <code>ps.penalty_newton_raphson(pEF)</code> <code>ps.frequency_analysis(p)</code>
<code>intForcCombFileName</code>	name of the file where are to be found the shell element forces and moments obtained for each ULS combination, to be used in the verification of the ULS due to shear. Directory path, file name and extension must be specified. The file contains a list of the following items (see the positive directions of the forces and moments in figure 7.3): $ULS_{id} \quad elem_{id} \quad N_1 \quad N_2 \quad N_{12} \quad M_1 M_2 \quad M_{12} \quad Q_1 \quad Q_2$
<code>outputFileName</code>	name of the output files (directory path and file name without extension) where to write the results of the verification
<code>mapSectionsForEveryElement</code>	data structure, such that, for each shell element: <code>mapSectionsForEveryElement[tagElem]=[nmbScc1,nmbScc2]</code> , where <code>nmbScc1</code> and <code>nmbScc2</code> are the names identifying the sections to be used in 1 and 2 directions respectively
<code>mapSectionsDefinition</code>	data structure that links each section name and direction with the respective record that contains the section definition parameters
<code>mapInteractionDiagrams</code>	data structure that links each section name and direction with the interaction diagram object to be used for its verification
<code>procesResultVerifV</code>	name of the function to be used for carrying out the section verification, e.g. <code>shearSIA262.procesResultVerifV</code>

#### 7.3.1.4 Cracking limit state verification

```
from materials.xLamina import calculo_fis
calculo_fis.lanzaCalculoFISFromXCDataPlanB(preprocessor,analysis,intForcCombFileName,
outputFileName, mapSectionsForEveryElement,mapSectionsDefinition,
procesResultVerifFIS)
```

<code>preprocessor</code>	preprocessor name
<code>analysis</code>	type of analysis to be performed. Some predefined types can be selected: <code>from solution import predefined_solutions as ps</code> <code>pEF=xc.ProblemaEF()</code> <code>ps.simple_static_linear(pEF)</code> for linear static analysis <code>ps.simple_newton_raphson(pEF)</code> for non linear analysis using Newton Raphson algorithm to solve the equation

	<pre> ps.simple_newton_raphson_band_gen(pEF) ps.simple_static_modified_newton(pEF) for non linear analysis using the modified Newton Raphson algorithm to solve the equation ps.penalty_newton_raphson(pEF) ps.frequency_analysis(p) </pre>
intForcCombFileName	name of the file where are to be found the shell element forces and moments obtained for each quasi permanent SLS or frequent SLS combination, to be used in the verification of the cracking limit state. Directory path, file name and extension must be specified. The file contains a list of the following items (see the positive directions of the forces and moments in figure 7.3): $ULS_{id} \quad elem_{id} \quad N_1 \quad N_2 \quad N_{12} \quad M_1 M_2 \quad M_{12} \quad Q_1 \quad Q_2$
outputFileName	name of the output files (directory path and file name without extension) where to write the results of the verification
mapSectionsForEveryElement	data structure, such that, for each shell element: <code>mapSectionsForEveryElement[tagElem]=[nmbScc1,nmbScc2]</code> , where <code>nmbScc1</code> and <code>nmbScc2</code> are the names identifying the sections to be used in 1 and 2 directions respectively
mapSectionsDefinition	data structure that links each section name and direction with the respective record that contains the section definition parameters
mapInteractionDiagrams	data structure that links each section name and direction with the interaction diagram object to be used for its verification
procesResultVerifFIS	name of the function to be used for carrying out the section verification, e.g. <code>materi-als.sia262.crackControlSIA262.procesResultVerifFISSIA262PlanB</code>

## 7.3.2 Verification of beam sections

## 7.3.3 Punching shear calculation

### 7.3.3.1 Punching shear calculation according to EC2

## Chapter 8

# Rough calculations

### 8.1 Punching shear

```
import rough_calculations.ng_punzonamiento as punch

punch.esfuerzoPunzonamiento(qk,A)
    rough estimation of the load on the slab over a support (HL.3 num. gordos)
punch.punzMaximo(fck,d,a,b)
    rough estimation of the maximum punching force with no need
    of reinforcement for punching shear (HL.3 num. gordos)
punch.reinforcementPunz(Vd,fck,d,a,b,h,fyd)
    rough estimation of the reinforcement for punching shear (HL.3 num. gordos)
```

where:

qk	characteristic uniformly distributed load on the slab
A	slab surface covered by the support
fck	characteristic compressive strength of concrete
d	effective depth of the slab
a,b	section dimensions of the support
h	overall depth of the slab
fydof the rein- forcement steel	design value of the yield strength

#### 8.1.1 Beam deflections

```
import rough_calculations.flechas_vigas as beamDefl
```

`beamDefl.deflCantBeamPconcentr(l,EI,P,a)`  
maximum deflection in a cantilever beam with a concentrated load at any point

`beamDefl.deflCantBeamQunif(l,EI,q)`  
maximum deflection in a cantilever beam with a uniformly distributed load

`beamDefl.deflCantBeamMend(l,EI,M)`  
maximum deflection in a cantilever beam with a couple moment at the free end

`beamDefl.deflSimplSupBeamPconcentr(l,EI,P,b)`  
maximum deflection in a beam simply supported at ends with a concentrated load at any point

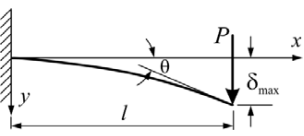
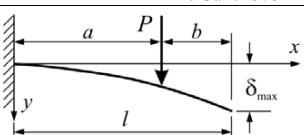
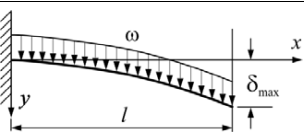
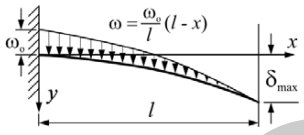
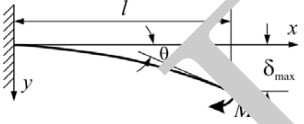
`beamDefl.deflSimplSupBeamQunif(l,EI,q)`  
maximum deflection in a beam simply supported at ends with a uniformly distributed load

`beamDefl.deflSimplSupBeamMend(l,EI,M)`  
maximum deflection in a beam simply supported at ends with a couple moment at the right end

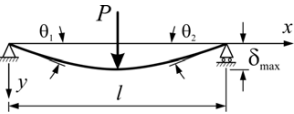
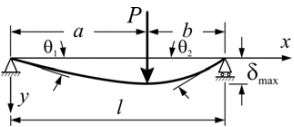
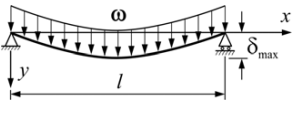
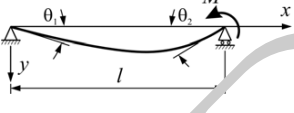
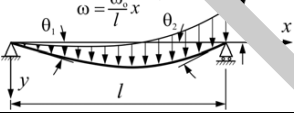
`l` span  
`EI` flexural rigidity of the section  
`P` concentrated load  
`q` uniformly distributed load  
`M` coupled moment  
`a` distance from the left end of the beam  
`b` distance from the right end of the beam



### BEAM DEFLECTION FORMULAE

BEAM TYPE	SLOPE AT FREE END	DEFLECTION AT ANY SECTION IN TERMS OF $x$	MAXIMUM DEFLECTION
1. Cantilever Beam – Concentrated load $P$ at the free end			
	$\theta = \frac{Pl^2}{2EI}$	$y = \frac{Px^2}{6EI}(3l-x)$	$\delta_{\max} = \frac{Pl^3}{3EI}$
2. Cantilever Beam – Concentrated load $P$ at any point			
	$\theta = \frac{Pa^2}{2EI}$	$y = \frac{Px^2}{6EI}(3a-x)$ for $x < a$ $y = \frac{Pa^2}{6EI}(3x-a)$ for $a < x < l$	$\delta_{\max} = \frac{Pa^2}{6EI}(3l-a)$
3. Cantilever Beam – Uniformly distributed load $\omega$ (N/m)			
	$\theta = \frac{\omega l^3}{6EI}$	$y = \frac{\omega x^2}{24EI}(x^2 + 6l^2 - 4lx)$	$\delta_{\max} = \frac{\omega l^4}{8EI}$
4. Cantilever Beam – Uniformly varying load (Maximum intensity $\omega_0$ (N/m))			
	$\theta = \frac{\omega_0 l^3}{24EI}$	$y = \frac{\omega_0 x^2}{20EI}(10l^3 - 10l^2x + 5lx^2 - x^3)$	$\delta_{\max} = \frac{\omega_0 l^4}{30EI}$
5. Cantilever Beam – Couple moment $M$ at the free end			
	$\theta = \frac{Ml}{EI}$	$y = \frac{Mx^2}{2EI}$	$\delta_{\max} = \frac{Ml^2}{2EI}$

### BEAM DEFLECTION FORMULAS

BEAM TYPE	SLOPE AT ENDS	DEFLECTION AT ANY SECTION IN TERMS OF $x$	MAXIMUM AND CENTER DEFLECTION
6. Beam Simply Supported at Ends – Concentrated load $P$ at the center			
	$\theta_1 = \theta_2 = \frac{Pl^2}{16EI}$	$y = \frac{Px}{12EI} \left( \frac{3l^2}{4} - x^2 \right)$ for $0 < x < \frac{l}{2}$	$\delta_{\max} = \frac{Pl^3}{48EI}$
7. Beam Simply Supported at Ends – Concentrated load $P$ at any point			
	$\theta_1 = \frac{Pb(l^2 - b^2)}{6EI}$ $\theta_2 = \frac{Pab(2l - b)}{6EI}$	$y = \frac{Pbx}{6EI} (l^2 - x^2 - b^2)$ for $0 < x < a$ $y = \frac{Pb}{6EI} \left[ \frac{l}{b} (x - a)^3 + (l^2 - b^2)x - \frac{3}{2}al^2 \right]$ for $a < x < l$	$\delta_{\max} = \frac{Pb(l^2 - b^2)^{3/2}}{9\sqrt{3}EI}$ at $x = \sqrt{(l^2 - b^2)}/3$ $\delta = \frac{Pb}{48EI} (3l^2 - 4b^2)$ at the center, if $a > b$
8. Beam Simply Supported at Ends – Uniformly distributed load $\omega$ (N/m)			
	$\theta_1 = \theta_2 = \frac{\omega l^3}{24EI}$	$y = \frac{\omega x}{24EI} (l^3 - 2lx^2 + x^3)$	$\delta_{\max} = \frac{5\omega l^4}{384EI}$
9. Beam Simply Supported at Ends – Couple $M$ at the right end			
	$\theta_1 = \frac{Ml}{6EI}$ $\theta_2 = \frac{Ml}{3EI}$	$y = \frac{Mlx}{6EI} \left( 1 - \frac{x^2}{l^2} \right)$	$\delta_{\max} = \frac{Ml^2}{9\sqrt{3}EI}$ at $x = \frac{l}{\sqrt{3}}$ $\delta = \frac{Ml^2}{16EI}$ at the center
10. Beam Simply Supported at Ends – Uniformly varying load: Maximum intensity $\omega_0$ (N/m)			
	$\theta_1 = \frac{7\omega_0 l^3}{36EI}$ $\theta_2 = \frac{2\omega_0 l^3}{45EI}$	$y = \frac{\omega_0 x}{360EI} (7l^4 - 10l^2 x^2 + 3x^4)$	$\delta_{\max} = 0.00652 \frac{\omega_0 l^4}{EI}$ at $x = 0.519l$ $\delta = 0.00651 \frac{\omega_0 l^4}{EI}$ at the center

## 8.2 Masonry bridge

Functions for the assessment of a masonry arch bridge by means of a deterministic analytical method.

In these routines, the mechanism method programmed is used for a single-span masonry in a limit analysis, giving a load carrying capacity and a failure mode of the structure. The algorithm of the method applies the kinematic approach. It uses the assumption that a masonry arch becomes a mechanism when at least four plastic hinges appears in the arch barrel. A strict and efficient calculation for unknown position of hinges is based on application of a linear programming algorithm with a target function minimising a live load factor. In this way the most probable mechanism mode is found automatically.

## 8.3 Soil thrust

DRAFT

## Appendix A

# Generation of combinations to consider in the structural calculation

### A.1 Introduction

This Appendix has the object of defining the actions, weighting coefficients and the combination of actions which shall be taken into account when designing structures.

Checking the structures through design is the most used method to guarantee their safety <sup>1</sup>.

#### A.1.1 The Limit States design method

The usual method prescribed by the codes for checking the safety of a structure is the so-called *Method of limit states*. A *limit state* is a situation in which, when exceeded, it may be considered that the structure does not fulfil one of the functions for which it has been designed.

The limit states are classified in:

- *Ultimate Limit States (ULS)*;
- *Serviceability Limit States (SLS)*, and
- *Durability Limit States (DLS)*.

#### A.1.2 Design situations

The concept of *design situation* is useful to sort the checks performed on the project or study of a structure. A design situation is a simplified representation of the reality that is amenable to analysis.

Thus, it can be considered design situations those that correspond to the different phases of construction of the structure, the normal use of the structure, its reparation, exceptional conditions, ....

For each of the design situations, it must be checked that the structure doesn't exceed any of the Limit States previously laid down in paragraph A.1.1

---

<sup>1</sup>Other procedures are also acceptable such as the reduced model tests, full-scale tests of the structure or its elements, extrapolation of the behaviour of similar structures, ...

Type of structure	Design working life
Temporary structures (*)	3 to 10 years (*)
Replaceable structural elements that are not part of the main structure (eg, handrails, pipe supports)	10 to 25 years
Agricultural or industrial buildings (or installations) and maritime works	15 to 50 years
Residential buildings or offices, bridges or crossings of a total length of less than 10 meters and civil engineering structures (except maritime works) having a low or average economic impact	50 years
Public buildings, health and education.	75 years
Monumental buildings or having a special importance	100 years
Bridges of total length equal to or greater than 10 meters and other civil engineering structures of high economic impact	100 years
(*)In accordance with the purpose of the structure (temporary exposure, etc.). Under no circumstances shall structures with a design working life greater than 10 years be regarded as temporary structures.	

Table A.1: Design working life of the various types of structure (according reference [4]).

### A.1.3 Actions

*Action* is defined as any cause capable of producing stress states in a structure, or modifying the existing one. Weight coefficients can be different according to the codes that apply for verification of the different structural elements (IAP, EHE, Eurocodes,...).

### A.1.4 Working life

The working life of a structure is the period of time from the end of its execution, during which must maintain the requirements of security and functionality of project and an acceptable aesthetic appearance. During that period it will require conservation in accordance with the maintenance plan established for that purpose.

The design working life depends on the type of structure and must be fixed by the Owners at the start of the design. In any case its duration will be lower than that indicated in the regulations applicable or, in the absence of these, than the values laid down in Table A.1.

When a structure consists of different members, different working life values may be adopted for such members, always in accordance with the type and characteristics of the construction thereof.

### A.1.5 Risk level

The level of risk of an infrastructure defines the consequences of a structural failure during its construction or service (public building, private store, bridge, ...)

### A.1.6 Control level

Regardless of the rigor with which the checking calculations of the structure are made during the project, its safety also depends on careful construction of it. Different standards establish the influence that the level of control during the execution of the work has on safety factors to be used in the execution of the same.

### A.1.7 Combination of actions

When designing a structure or a structural member by the limit state method, load combinations shall be considered as the sum of the products of the load effect corresponding to the basic value of each load and the load factor.

Load factors shall be determined appropriately considering the limit state, the target reliability index, the variability in the load effect of each load and resistance, the probability of load coincidence, etc.

### A.1.8 Verification of the structure

From the discussion in the previous sections, the verification procedure of the structure will consist of performing the following tasks:

1. identify the design situations to be considered when checking the structure;
2. identify the load criteria hypotheses for each of those design situations;
3. define the combinations of actions to be considered when checking the ULS and SLS, depending on:
  - (a) materials composing the structure or the element to check: rolled steel, reinforced concrete, wood, ...;
  - (b) risk level of the infrastructure
  - (c) level of control with which the construction work is performed;
  - (d) design situation (persistent, transient or accidental)
4. obtain the calculation value of the effect of actions for each combination.
5. verify all the limit states.

## A.2 Actions

An action is a set of forces applied to the structure or a set of imposed deformations or accelerations, that has an effect on structural members (e.g. internal force, moment, stress, strain) or on the whole structure (e.g. deflection, rotation)

### A.2.1 Classification of actions

Actions can be classified by their variation over time, their nature, their origin, their spatial variation, ...

#### A.2.1.1 By their nature

- **Direct actions:** loads applied to the structure (e.g. self-weight, dead load, live load, ...)
- **Indirect actions:** imposed deformations or accelerations caused for example by temperature changes, moisture variation,...

#### A.2.1.2 By their variation over time

Actions shall be classified by their variation in time, by reference to their *service life*<sup>2</sup>, as follows:

- **Permanent actions G:** actions that are likely to act throughout a given reference period and for which the variation in magnitude with time is negligible, or for which the variation is always in the same direction (monotonic) until the action attains a certain limit value, e.g. self-weight of structures, fixed equipment and road surfacing, and indirect actions caused by shrinkage and uneven settlement.
- **Permanents of a non-constant value G\*:** are those which act at any time but whose magnitude is non constant. This group include those actions whose variation is a function of elapsed time and are produced in a single direction, tending towards a certain limit value (rheological actions, pretensioning, subsidence of the ground under the foundations, ...). They also include other actions originating from the ground whose magnitude does not vary as a function of time but as a function of the interaction between the ground and the structure (for example, thrusts on vertical elements).
- **Variables Q:** action for which the variation in magnitude with time is neither negligible nor monotonic. E.g. imposed loads on building floors, beams and roofs, wind actions or snow loads.
- **Accidental actions A:** action, usually of short duration but of significant magnitude, that is unlikely to occur on a given structure during the design working life. E.g. explosions, or impact from vehicles.
- **seismic action AS:** action that arises due to earthquake ground motions.

#### A.2.1.3 By their origin

- **Gravitational:** which has its origin in the earth's gravitational field (self-weight, dead load, earth pressure, ...)
- **Climatic:** whose origin is in the climate (thermal action and wind actions<sup>3</sup>)
- **Rheological:** which has its origin in the response of material with plastic flow rather than deforming elastically when a force is applied (e.g. shrinkage of concrete).
- **Seismic:** due to earthquake ground motions.

<sup>2</sup>See section A.1.4.

<sup>3</sup>thermal and wind actions can not be due to climate, such as in the case of an oven or structures subjected to the thrust of jet engines of aircraft



**A.2.1.4 By the structural response which they produce**

- **static action:** action that does not cause significant acceleration of the structure or structural members;
- **dynamic action:** action that causes significant acceleration of the structure or structural members;
- **quasi-static action:** dynamic action represented by an equivalent static action in a static model.

**A.2.1.5 By their spatial variation**

- **fixed action:** action that has a fixed distribution and position over the structure or structural member such that the magnitude and direction of the action are determined unambiguously for the whole structure or structural member if this magnitude and direction are determined at one point on the structure or structural member;
- **free action:** action that may have various spatial distributions over the structure.

**A.2.1.6 By their relation with other actions**

- **Compatible actions:** two actions are compatible when it's possible for them to act simultaneously.
- **Incompatible actions:** two actions are incompatible when it's impossible for them to act at the same time (e.g. one crane acting simultaneously in two different positions).
- **Synchronous actions:** two actions are synchronous when they act necessarily together, at the same time (e.g. the braking load of a crane bridge will be synchronised with the action of the weight of the crane).

**A.2.1.7 By their participation in a combination**

- **Leading action:** in a combination of actions, the leading variable action is the one which produces the largest design load effect; its characteristic value is used.
- **Accompanying action:** variable action that accompanies the leading action in a combination; its characteristic value is reduced by using a factor  $\Psi$ .

**A.2.2 Values of actions****A.2.2.1 Characteristic value of an action  $F_k$** 

It is the principal representative value of an action; it is chosen so as to correspond to a 5% probability of not being exceeded on the unfavourable side during a "reference period" taking into account the design working life of the structure and the duration of the design situation.

**A.2.2.2 Combination value of a variable action  $F_{r0}$** 

Value chosen so that the probability that the effects caused by the combination will be exceeded is approximately the same as by the characteristic value of an individual action. It may be expressed as a determined part of the characteristic value by using a factor  $\Psi_0 \leq 1$

### A.2.2.3 Frequent value of a variable action $F_{r1}$

Value determined so that either the total time, within the reference period, during which it is exceeded is only a small given part of the reference period, or the frequency of it being exceeded is limited to a given value. It may be expressed as a determined part of the characteristic value by using a factor  $\Psi_1 \leq 1$ .

### A.2.2.4 Quasi-permanent value of a variable action $F_{r2}$

Value determined so that the total period of time for which it will be exceeded is a large fraction <sup>4</sup> of the reference period. It may be expressed as a determined part of the characteristic value by using a factor  $\Psi_1 \leq 2$ .

### A.2.2.5 Representative value $F_r$ of the actions. Factors of simultaneity

The representative value of an action is the value of it that is used to verify the limit states. By multiplying this representative value by the the corresponding partial coefficient  $\gamma_f$ , the calculation value shall be obtained.

The principal representative value of the actions is their characteristic value. Usually, for permanent and accidental actions, a single representative value is considered, that matches the characteristic value ( $\psi = 1$ ) <sup>5</sup>. Other representative values are considered for the variable actions, in accordance with the verification involved and the type of action:

- **Characteristic value**  $F_k$ : this value is used for leading actions in the verification of ultimate limit states in a continuous or temporary situation and of irreversible serviceability limit states.
- **Combination value**  $\psi = \psi_0 F_k$  this value is used for accompanying actions in the verification of ultimate limit states in a continuous or temporary situation and of irreversible serviceability limit states.
- **Frequent value**  $\psi = \psi_1 F_k$ : this value is used for the leading action in the verification of ultimate limit states in an accidental situations and of reversible serviceability limit states.
- **Quasi-permanent value**  $\psi = \psi_2 F_k$ : this value is used for accompanying actions in the verification of ultimate limit states in an accidental situation and of reversible serviceability limit states as well as in the assessment of the postponed effects.

In short, the representative value of an action depends on:

- its variation over time (G,G\*,Q,A,AS);
- its participation in the combination as *leading action* or *accompanying action*;
- the type of situation (accidental, ...);
- the origin of the load (climate, use, water, ...).

**A.2.2.5.1 Values of  $\Psi$  factors of simultaneity** The value of the simultaneity factors  $\psi$  are different depending on the action that is involved.

<sup>4</sup>according to *Documento Nacional de Aplicación español del Eurocódigo de Hormigón (UNE ENV 1992-1-1)* more than half of the service life of the structure

<sup>5</sup>The IAP instruction (reference [5]) makes some exceptions to this rule)

CLIMATIC ACTIONS	$\psi_0$	$\psi_1$	$\psi_2$
Snow loads	0.6	0.2	0.0
Wind loads	0.6	0.5	0.0
Temperature ( <i>non-fire</i> )	0.6	0.5	0.0

Table A.2: Recommended values of  $\Psi$  factor for climatic actions, according to EHE

**According to EHE:** the recommended values of factors of simultaneity  $\psi_0, \psi_1, \psi_2$  according to the *Documento Nacional de Aplicación español del Eurocódigo de Hormigón* (UNE ENV 1992-1-1) can be seen in tables A.2 y A.3.

**According to EAE [4] :** see tables A.5 y A.4.

**According to IAP [5]:** see table A.6.

#### A.2.2.6 Calculation value $F_d$ of the actions

The calculation value of an action is obtained by multiplying its characteristic value by the corresponding partial coefficient  $\gamma_f$ :

$$F_d = \gamma_f \cdot F_r \quad (\text{A.1})$$

The values of the coefficients  $\gamma_f$  takes into account one or more of the following uncertainties:

1. uncertainties in the estimation of the representative value of the actions, in fact, the characteristic value is chosen admitting a 5% probability of being exceeded during the working life of the structure;
2. uncertainties in the calculations results, due to simplifications in the models and to certain numeric errors (rounding, truncation, ...)
3. Uncertainty in the geometric and mechanical characteristics of the built structure. During the execution of the structure some errors can be committed <sup>6</sup> that can make the dimensions of the sections, the position of the reinforcement, the position of the axes, the mechanical characteristics of the materials, ..., be different from the theoretical.

**A.2.2.6.1 Values of the partial coefficients** The coefficients  $\gamma_f$  have different values in accordance with:

1. the limit state to be verified;
2. the design situation that is involved (see section A.3);
3. the variation of the action over time (according to classification in A.2.1.2);
4. the effect favourable o unfavourable of the action in the limit state that is verified;
5. the control level.

**According to EHE:** the values of the partial coefficients  $\gamma_f$  are specified in table A.7 for serviceability limit states and in table A.8 for ultimate limit states.

<sup>6</sup>It is understood that these errors are within the tolerances established in the regulations

LIVE LOADS	$\psi_0$	$\psi_1$	$\psi_2$
<b>Roofs</b>			
Inaccessible or accessible only for maintenance	0.7	0.5	0.3
Accessible	by use	by use	by use
<b>Residential buildings</b>			
Rooms	0.7	0.5	0.3
Stairs and public accesses	0.7	0.5	0.3
Cantilevered balconies	0.7	0.5	0.3
<b>Hotels, hospitals, prisons, ...</b>			
Bedrooms	0.7	0.5	0.3
Public areas, stairs and accesses	0.7	0.7	0.6
Assembly and areas	0.7	0.7	0.6
Cantilevered balconies	by use	by use	by use
<b>Office and commercial buildings</b>			
Private premises	0.7	0.5	0.3
Public offices	0.7	0.5	0.3
Shops	0.7	0.7	0.6
Commercial galleries, stairs and access	0.7	0.7	0.6
Storerooms	1.0	0.9	0.8
Cantilevered balconies	by use	by use	by use
<b>Educational buildings</b>			
Classrooms, offices and canteens	0.7	0.7	0.6
Stairs and access	0.7	0.5	0.6
Cantilevered balconies	by use	by use	by use
<b>Churches, buildings for assembly and public performances</b>			
Halls with fixed seatings	0.7	0.7	0.6
Halls without fixed seatings, tribunes, stairs	0.7	0.7	0.6
Cantilevered balconies	by use	by use	by use
<b>Driveways and garages</b>			
Traffic areas with vehicles under 30 kN in weight	0.7	0.7	0.6
Traffic areas with vehicles of 30 to 160 kN in weight	0.7	0.5	0.3

Table A.3: Recommended values of  $\Psi$  factors of simultaneity for climatic loads, according to EHE

USE OF AREA	$\psi_0$	$\psi_1$	$\psi_2$
Domestic, residential areas	0.7	0.5	0.3
Office areas	0.7	0.5	0.3
Congregation areas	0.7	0.7	0.6
Shopping areas	0.7	0.7	0.6
Storage areas	1.0	0.9	0.8
Traffic areas, weight of vehicle $\leq 30$ kN	0.7	0.7	0.6
Traffic areas, $30$ kN < weight of vehicle $\leq 160$ kN	0.7	0.5	0.3
Inaccessible Roofs	0.0	0.0	0.0

Table A.4: Recommended values of  $\Psi$  factors for buildings, according to EAE

CLIMATIC ACTIONS	$\psi_0$	$\psi_1$	$\psi_2$
Snow loads in buildings set over a thousand meters above sea level.	0.7	0.5	0.2
Snow loads in buildings set under a thousand meters above sea level.	0.5	0.2	0.0
Wind loads	0.6	0.2	0.0
Thermal action	0.6	0.5	0.0

Table A.5: Recommended values of  $\Psi$  factors of simultaneity, according to EAE

VARIABLE ACTIONS	$\psi_0$	$\psi_1$	$\psi_2$
Traffic load model fatigue	1.0	1.0	1.0
Other variable actions	0.6	0.5	0.2

Table A.6: Values of  $\Psi$  factors of simultaneity according to IAP.

ACTION	EFFECT	
	favourable	unfavourable
Permanent	$\gamma_G = 1.00$	$\gamma_G = 1.00$
Prestressing (pre-tensioned concrete)	$\gamma_P = 0.95$	$\gamma_P = 1.05$
Prestressing (post-tensioned concrete)	$\gamma_P = 0.90$	$\gamma_P = 1.10$
Permanent of a non-constant value	$\gamma_{G*} = 1.00$	$\gamma_{G*} = 1.00$
Variable	$\gamma_Q = 0.00$	$\gamma_Q = 1.00$
NOTATION:		
G: Permanent action.		
P: Prestressing.		
G*: Permanent action of a non-constant value.		
Q: Variable action.		
A: Accidental action.		

Table A.7: Partial factor for actions in serviceability limit states according to EHE.

Action	Control level	Effect in persistent or transient design situations		Effect in accidental or seismic design situations	
		favourable	unfavourable	favourable	unfavourable
G	intense	$\gamma_G = 1.00$	$\gamma_G = 1.35$	$\gamma_G = 1.00$	$\gamma_G = 1.00$
	normal	$\gamma_G = 1.00$	$\gamma_G = 1.50$	$\gamma_G = 1.00$	$\gamma_G = 1.00$
	low	$\gamma_G = 1.00$	$\gamma_G = 1.60$	$\gamma_G = 1.00$	$\gamma_G = 1.00$
G*	intense	$\gamma_{G*} = 1.00$	$\gamma_{G*} = 1.50$	$\gamma_{G*} = 1.00$	$\gamma_{G*} = 1.00$
	normal	$\gamma_{G*} = 1.00$	$\gamma_{G*} = 1.60$	$\gamma_{G*} = 1.00$	$\gamma_{G*} = 1.00$
	low	$\gamma_{G*} = 1.00$	$\gamma_{G*} = 1.80$	$\gamma_{G*} = 1.00$	$\gamma_{G*} = 1.00$
Q	intense	$\gamma_Q = 0.00$	$\gamma_Q = 1.50$	$\gamma_Q = 0.00$	$\gamma_Q = 1.00$
	normal	$\gamma_Q = 0.00$	$\gamma_Q = 1.60$	$\gamma_Q = 0.00$	$\gamma_Q = 1.00$
	low	$\gamma_Q = 0.00$	$\gamma_Q = 1.80$	$\gamma_Q = 0.00$	$\gamma_Q = 1.00$
A	-	-	-	$\gamma_A = 1.00$	$\gamma_A = 1.00$
NOTATION:					
G: Permanent action.					
G*: Permanent action of a non-constant value.					
Q: Variable action.					
A: Accidental action.					

Table A.8: Partial factor for actions in ultimate limit states according to EHE.

ACTION	EFFECT	
	favourable	unfavourable
Permanent	$\gamma_G = 1.00$	$\gamma_G = 1.00$
Permanent of a non-constant value	$\gamma_{G*} = 1.00$	$\gamma_{G*} = 1.00$
Variable	$\gamma_Q = 0.00$	$\gamma_Q = 1.00$

Table A.9: Partial factor for actions in serviceability limit states according to EAE.

Action	Effect in persistent or transient design situations		Effect in accidental or seismic design situations	
	favourable	unfavourable	favourable	unfavourable
G	$\gamma_G = 1.00$	$\gamma_G = 1.35$	$\gamma_G = 1.00$	$\gamma_G = 1.00$
G*	$\gamma_{G*} = 1.00$	$\gamma_{G*} = 1.50$	$\gamma_{G*} = 1.00$	$\gamma_{G*} = 1.00$
Q	$\gamma_Q = 0.00$	$\gamma_Q = 1.50$	$\gamma_Q = 0.00$	$\gamma_Q = 1.00$
A	-	-	$\gamma_A = 1.00$	$\gamma_A = 1.00$
NOTATION:				
G: Permanent action.				
G*: Permanent action of a non-constant value.				
Q: Variable action.				
A: Accidental action.				

Table A.10: Partial factor for actions in ultimate limit states according to EAE.

**According to EAE:** the values of the partial coefficients  $\gamma_F$  to be used are specified in tables A.9 for serviceability limit states and in table A.10 for ultimate limit states.

**According to IAP:** the values of the partial coefficients  $\gamma_F$  to be used are specified in tables A.11 for serviceability limit states and in table A.12 for ultimate limit states.

### A.3 Design situations

Design situations, that take into account the circumstances under which the structure can be required during its execution and use, shall be classified as follows:

1. Persistent design situations, which refer to the conditions of normal use.
2. transient design situations, which refer to temporary conditions applicable to the structure, e.g. during execution or repair.
3. Accidental design situations, which refer to exceptional conditions applicable to the structure or to its exposure, e.g. to fire, explosion, impact or the consequences of localised failure.

### A.4 Level of quality control

A two level system for control during execution has been adopted:

- Intense control.
- Normal control.

ACTION	EFFECT	
	favourable	unfavourable
Permanent	$\gamma_G = 1.00$	$\gamma_G = 1.00$
Internal prestressing (post-tensioned concrete)	$\gamma_{P_1} = 0.9$	$\gamma_{P_1} = 1.1$
Internal prestressing (pre-tensioned concrete)	$\gamma_{P_1} = 0.95$	$\gamma_{P_1} = 1.05$
External prestressing	$\gamma_{P_2} = 1.0$	$\gamma_{P_2} = 1.0$
Other prestressing actions	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.00$
Rheological	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.00$
Thrust of the site	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.00$
Variable	$\gamma_Q = 0.00$	$\gamma_Q = 1.00$
NOTATION:		
$G$ : Permanent action.		
$P_1$ : Internal prestressing.		
$P_2$ : External prestressing.		
$G^*$ : Permanent action of a non-constant value.		
$Q$ : Variable action.		
$A$ : Accidental action.		

Table A.11: Partial factor for actions in serviceability limit states according to IAP.

Action	Effect in persistent or transient design situations		Effect in accidental or seismic design situations	
	favourable	unfavourable	favourable	unfavourable
Permanent	$\gamma_G = 1.00$	$\gamma_G = 1.35$	$\gamma_G = 1.00$	$\gamma_G = 1.00$
Internal prestressing	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.00$
External prestressing	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.35$	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.00$
Other prestressing actions	$\gamma_{G^*} = 0.95$	$\gamma_{G^*} = 1.05$	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.00$
Rheological	$\gamma_{G^*} = 1.0$	$\gamma_{G^*} = 1.35$	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.00$
Thrust of the site	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.50$	$\gamma_{G^*} = 1.00$	$\gamma_{G^*} = 1.00$
Variable	$\gamma_Q = 0.00$	$\gamma_Q = 1.50$	$\gamma_Q = 0.00$	$\gamma_Q = 1.00$
Accidental	-	-	$\gamma_A = 1.00$	$\gamma_A = 1.00$

Table A.12: Partial factor for actions in ultimate limit states according to IAP.

As will be seen later, the partial factors for a material or a member resistance depend on the level of inspection during construction.

## A.5 Limit states

They can be defined as those states beyond which the structure no longer fulfils the relevant design criteria.

The design of the structure will be right when:

1. it is verified that no ultimate limit state is exceeded for the design situations and load cases defined in A.6.1, and
2. it is verified that no serviceability limit state is exceeded under the design situations and load cases defined in A.6.2.

### A.5.1 Ultimate limit states

They are states associated with collapse or with other similar forms of structural failure. They generally correspond to the maximum load-carrying resistance of a structure or structural member.

The following ultimate limit states shall be verified where they are relevant: - failure caused by fatigue or other time-dependent effects.

1. loss of equilibrium of the structure or any part of it, considered as a rigid body;
2. failure by excessive deformation, transformation of the structure or any part of it into a mechanism, rupture, loss of stability of the structure or any part of it, including supports and foundations;
3. failure caused by fatigue or other time-dependent effects.

### A.5.2 Serviceability limit states

They can be defined as states that correspond to conditions beyond which specified service requirements for a structure or structural member are no longer met. These service requirements can concern:

- functionality.
- comfort.
- durability.
- aesthetics.

The verification of serviceability limit states should be based on criteria concerning the following aspects :

1. deformations that affect:
  - the appearance,
  - the comfort of users, or



- the functioning of the structure (including the functioning of machines or services),
- or that cause damage to finishes or non-structural members;
- 2. vibrations
  - that cause discomfort to people, or
  - that limit the functional effectiveness of the structure;
- 3. damage that is likely to adversely affect
  - the appearance,
  - the durability, or
  - the functioning of the structure.

## A.6 Combination of actions

When the verification of a structure is carried out by the partial factor method, it shall be verified that, in all relevant design situations, no relevant limit state is exceeded when design values for actions or effects of actions and resistances are used in the design models.

In order to eliminate the combinations that are not possible (or do not make sense), the following criteria will be considered:

- When an action is involved in a combination, none of its incompatible actions will be involved in that combination.
- When an action is involved in a combination, all of its synchronous actions must be involved in that combination <sup>7</sup>

In what follows, we will consider any structure, under the following actions:

- $n_G$  permanent actions:  $G_i$  <sup>8</sup>.
- $n_{G*}$  permanent actions of a non-constant value:  $G*_j$ .
- $n_Q$  variable actions:  $Q_l$ .
- $n_A$  accidental actions:  $Q_m$ .
- $n_{AS}$  seismic actions:  $Q_n$ .

### A.6.1 Combinations of actions for ultimate limit states

For the selected design situations and the relevant ultimate limit states the individual actions for the critical load cases should be combined as detailed in this section.

<sup>7</sup>See synchronous action and compatible action definitions in section A.2.1.6.

<sup>8</sup>The subscript refers to each of permanent actions on the structure  $G_1, G_2, G_3, G_4, \dots, G_{n_G}$

### A.6.1.1 Combinations of actions for persistent or transient design situations

For each variable action, a group of combinations with this action as *leading variable action* will be considered <sup>9</sup>.

$$\sum_{i=1}^{n_G} \gamma_G \cdot G_{k,i} + \sum_{j=1}^{n_{G*}} \gamma_{G*} \cdot G_{*k,j} + \gamma_Q \cdot Q_{k,d} + \sum_{l=1}^{d-1} \gamma_Q \cdot Q_{r0,l} + \sum_{l=d+1}^{n_Q} \gamma_Q \cdot Q_{r0,l} \quad (\text{A.2})$$

where:

- $\gamma_G \cdot G_{k,i}$ : design value of the permanent action  $i$ , obtained from its characteristic value ;
- $\gamma_{G*} \cdot G_{*k,j}$ : design value of the permanent action of a non-constant value  $j$ , obtained from its characteristic value;
- $\gamma_Q \cdot Q_{k,d}$ : design value of the leading variable action  $d$ , obtained from its characteristic value;
- $\gamma_Q \cdot Q_{r0,l}$ : design value of la variable action  $l$ , obtained from its accompanying value.

#### A.6.1.1.1 Number of combinations to be considered: According to section A.2.2.6:

- The permanent actions, in ULS combinations corresponding to persistent or transient design situations, will have two non-zero partial factors.
- In the same case, the permanent actions of a non-constant value will have two non-zero partial factors that, in some cases, can be equal (see the case of internal prestressing on the table A.12).
- The variable actions will have a single non-zero partial factor.

therefore, assuming that:

$n_{G2}$  is the number of permanent actions that have two different partial factors;

$n_{G1}$  is the number of permanent actions that have a single partial factor<sup>10</sup>;

$n_{G*2}$  is the number of permanent actions of a non-constant value that have two different partial factors;

$n_{G*1}$  the number of permanent actions of a non-constant value that have a single partial factor, and

$n_Q$  is the number of variable actions, all of then have a single partial factor.

If, by now, incompatibility or synchronicity of actions is ignored, for each variable action we'll have:

- $2^{n_{G2}}$  combinations of permanent actions in the set  $G2$ ;
- 1 combination of permanent actions in the set  $G1$ ;
- $2^{n_{G*2}}$  combinations of permanent actions in the set  $G * 2$ ;

<sup>9</sup>See section A.2.1.7.

<sup>10</sup>Because both factors are equal.

- 1 combination of permanent actions in the set  $G * 1$ , and
- $2^{n_Q-1}$  combinations of accompanying variable actions.

As, for each leading action two partial factors must be considered, the total number of combinations  $n_{comb,spt}$  for persistent or transient design situations will be equal to the cartesian product of the previous combinations by  $2^{n_{Qd}}$ , where  $Qd$  is the number of variable actions that can be leading:

$$n_{comb,ULS,spt} = 2^{n_{G2}} \cdot 2^{n_{G*2}} \cdot 2^{n_Q-1} \cdot 2^{n_{Qd}} = 2^{n_{G2}+n_{G*2}+n_Q+n_{Qd}-1} \quad (A.3)$$

Among these combinations, those that are incompatibles must be eliminated.

For synchronic actions, the following procedure can be followed:

Let  $a$  be a synchronic action of the action  $b$ :

1.  $a$  is eliminated from the list of variable actions;
2. the action  $a + b$  is added to the list of variable actions;
3. incompatibility between  $a + b$  and  $b$  actions is set.

#### A.6.1.2 Combinations of actions for accidental design situations

For each variable action  $Q_l$ ,  $n_A$  combinations with that action as leading are formed.

$$\sum_{i=1}^{n_G} \gamma_G \cdot G_{k,i} + \sum_{j=1}^{n_{G*}} \gamma_{G*} \cdot G_{*k,j} + A_{k,m} + \gamma_Q \cdot Q_{r1,d} + \sum_{l=1}^{d-1} \gamma_Q \cdot Q_{r2,l} + \sum_{l=d+1}^{n_Q} \gamma_Q \cdot Q_{r2,l} \quad (A.4)$$

where:

$A_{k,m}$ : design value of the accidental action  $m$ , obtained from its characteristic value;

$\gamma_Q \cdot Q_{r1,d}$ : design value of the leading variable action  $d$ , obtained from its representative frequent value;

$\gamma_Q \cdot Q_{r2,l}$ : design value of the variable action  $l$ , obtained from its representative quasi-permanent value.

**A.6.1.2.1 Number of combinations to be considered:** it results the same number of combinations for each sum than in the case solved in the paragraph A.6.1.1 (see A.3 expression), though, in this case, the representative values of the variable actions are other ones. If, as usual, the partial factors for seismic actions are equal for favourable and unfavourable actions, it suffices to multiply by the number of accidental actions  $n_A$ .

$$n_{comb,ULS,acc} = 2^{n_{G2}+n_{G*2}+n_Q+n_{Qd}-1} \cdot n_A \quad (A.5)$$

For incompatible actions, the procedure provided for in section A.6.1.1 is applicable.

### A.6.1.3 Combinations of actions for seismic design situations

For each seismic action one combination will be formed:

$$\sum_{i=1}^{n_G} \gamma_G \cdot G_{k,i} + \sum_{j=1}^{n_{G*}} \gamma_{G*} \cdot G_{*k,j} + AS_{k,n} + \sum_{l=1}^{n_Q} \gamma_Q \cdot Q_{r2,l} \quad (\text{A.6})$$

where:

$A_{k,m}$  is the design value of the accidental action  $m$ , and

$\gamma_Q \cdot Q_{r2,l}$  is the design value of the variable action  $l$ , obtained from its representative quasi-permanent value.

#### A.6.1.3.1 Number of combinations to be considered:

$$n_{comb,ULS,ismic} = 2^{n_{G2}+n_{G*2}+n_Q} \cdot n_{AS} \quad (\text{A.7})$$

For incompatible actions, the procedure provided for in section A.6.1.1 is applicable.

## A.6.2 Combinations of actions for serviceability limit states

For the selected design situations and the relevant serviceability limit states the individual actions for the critical load cases should be combined as detailed in this section.

### A.6.2.1 Rare combinations:

For each variable action, one combination with this action as *leading variable action* will be considered.

$$\sum_{i=1}^{n_G} G_{k,i} + \sum_{j=1}^{n_{G*}} G_{*k,j} + Q_{k,d} + \sum_{l=1}^{d-1} Q_{r0,l} + \sum_{l=d+1}^{n_Q} Q_{r0,l} \quad (\text{A.8})$$

In a general case, with no incompatible or concomitant combinations, the following combinations will be formed (see notation in section A.6.1.1):

$$n_{comb,SLS,pf} = 2^{n_{G2}+n_{G*2}+n_Q+n_{Qd}-1} \quad (\text{A.9})$$

Since the partial factors are for serviceability limit states, the sets  $G2$  y  $G*2$  generally will not match those for ultimate limit states. Given that in many cases both partial factors are equal to the unity, the cardinality of these sets will be much lower than the equivalent in paragraph A.6.1.1.

For incompatible actions, the procedure provided for in section A.6.1.1 is applicable.

### A.6.2.2 Frequent combinations:

For each variable action, one combination in which this action acts as *leading* will be formed.

$$\sum_{i=1}^{n_G} G_{k,i} + \sum_{j=1}^{n_{G*}} G_{*k,j} + Q_{r1,d} + \sum_{l=1}^{d-1} Q_{r2,l} + \sum_{l=d+1}^{n_Q} Q_{r2,l} \quad (\text{A.10})$$

the number of combinations will be the same as the precedent case, since only the combination factors can vary.

### A.6.2.3 Quasi-permanent combinations:

$$\sum_{i=1}^{n_G} G_{k,i} + \sum_{j=1}^{n_{G*}} G_{*k,j} + \sum_{l=1}^{n_Q} Q_{r2,l} \quad (\text{A.11})$$

the number of combinations will be:

$$n_{comb,SLS,cp} = 2^{n_{G2}+n_{G*2}+n_Q} \quad (\text{A.12})$$

### A.6.3 Combinations to be considered in the calculation

According to the discussion in the previous sections, the number of combinations for a general calculations will be the following:

Ultimate limit states	number of combinations
Persistent or transient design situations	$2^{(n_G+n_{G*}+n_Q)} \cdot n_Q$
Accidental design situations	$2^{(n_G+n_{G*}+n_Q)} \cdot n_Q \cdot n_A$
Seismic design situations	$2^{(n_G+n_{G*}+n_Q)} \cdot n_{AS}$
Total ULS	$2^{(n_G+n_{G*}+n_Q)} \cdot (n_Q(1+n_A) + n_{AS})$
Serviceability limit states	
Rare combinations	$n_Q$
Frequent combinations	$n_Q$
Quasi-permanent combination	1
Total SLS	$2n_Q + 1$
<b>Total combinations</b>	<b><math>2^{(n_G+n_{G*}+n_Q)} \cdot (n_Q(1+n_A) + n_{AS}) + 2n_Q + 1</math></b>

For example, if we had:

- 2 permanent actions;
- 1 permanent action of a non-constant value;
- 3 variable actions;
- 1 accidental action, and
- 2 seismic actions

the number of combinations will be:

Ultimate limit states	number of combinations
Persistent or transient design situations	$2^{(2+1+3)} \times 3 = 192$
Accidental design situations	$2^{(2+1+3)} \times 3 \times 1 = 192$
Seismic design situations	$2^{(2+1+3)} \times 2 = 128$
Total ULS	$2^{(2+1+3)} \times (3 \times (1+1) + 2) = 512$
Serviceability limit states	
Rare combinations	3
Frequent combinations	3
Quasi-permanent combination	1
Total SLS	$6 + 1 = 7$
<b>Total combinations</b>	<b>519</b>

### A.6.4 Algorithm to write the complete list of combinations

#### A.6.4.1 Combinations for ultimate limit states

Each of the sums in expressions (A.2),(A.4) y (A.6) appears as follows:

$$\sum_{i=1}^n \gamma_f \cdot F_{r,i} \quad (\text{A.13})$$

For each action  $F_i$  the partial factor can take two values, depending on the effect favourable or unfavourable of the action<sup>11</sup>.

The design value of the action  $F_{r,i}$  depends on:

- its variation in time (G,G\*,A,A,AS);
- its role in the combination, as leading or accompanying action;
- if there is or not accidental actions in the combination;
- the nature of the action (climatic or live loads).

in any case, for any combination, the value of  $F_{r,i}$  is known.

Moreover, the value of  $n$  is known for each sum.

Following this, the summands of (A.13) correspond to the variations with repetition<sup>12</sup> of two elements<sup>13</sup> taken  $n$  by  $n$ .

To write the variations with repetition of expression (A.13), proceed as follows:

Let  $\gamma_{\mathbf{f}_v}$  be the row vector whose components are the partial factors of the variation  $v$  ( $1 \leq v \leq 2^n$ ):

$$\gamma_{\mathbf{f}_v} = [\gamma_{f,1}, \gamma_{f,2}, \dots, \gamma_{f,i}, \dots, \gamma_{f,n}] \quad (\text{A.14})$$

that's to say, the element  $\gamma_{f,i}$  is the partial factor (favourable or unfavourable) of action  $F_{r,i}$ .

Let  $\mathbf{F}_r$  be the column vector whose components are the actions  $F_{r,i}$  of the expression (A.13):

$$\mathbf{F}_r^T = [F_{r,1}, F_{r,2}, \dots, F_{r,i}, \dots, F_{r,n}] \quad (\text{A.15})$$

then, the expression (A.13) is equivalent to the scalar product:

$$\sum_{i=1}^n \gamma_f \cdot F_{r,i} = \gamma_{\mathbf{f}_v} \cdot \mathbf{F}_r \quad (\text{A.16})$$

and it must be formed as many scalar products as variations with repetition can be arranged, that's to say,  $2^n$ .

Let  $S_{F,v}$  be the sum that corresponds to variation  $v$ ,

$$S_{F,v} = \gamma_{\mathbf{f}_v} \cdot \mathbf{F}_r \quad (\text{A.17})$$

then each of sums (A.2),(A.4) and (A.6) gives rise to set of variations:

<sup>11</sup>We assume a priori unknown the effect favourable or unfavourable of the action for the limit state and structural element in analysed

<sup>12</sup>The variations with repetition of  $n$  elements taken  $k$  by  $k$  are the arranged groups formed by  $k$  elements from  $A$  (which may be repeated)

<sup>13</sup>The partial factors corresponding to favourable and unfavourable effects

$$\begin{aligned}
 S_{F_r,1} &= \gamma_{f_1} \cdot \mathbf{F}_r \\
 S_{F_r,2} &= \gamma_{f_2} \cdot \mathbf{F}_r \\
 &\dots \\
 S_{F_r,v} &= \gamma_{f_v} \cdot \mathbf{F}_r \\
 &\dots \\
 S_{F_r,n_F} &= \gamma_{f_{n_F}} \cdot \mathbf{F}_r
 \end{aligned}$$

where  $n_F$  is the number of actions in each case, that's to say  $n_G$ ,  $n_{G*}$ ,  $n_Q$ ,  $n_A$ , or  $n_{AS}$ . Therefore, the summands (A.2),(A.4) and (A.6) will be one of the following scalar products:

- Summand corresponding to permanent actions:  $S_{G_r,v_G}$  ( $1 \leq v_G \leq 2^{n_G}$ ).
- Summand corresponding to permanent actions of a non-constant value:  $S_{G_{*r},v_{G*}}$  ( $1 \leq v_{G*} \leq 2^{n_{G*}}$ ).
- Summand corresponding to variable actions:  $S_{Q_r,v_Q}$  ( $1 \leq v_Q \leq 2^{n_Q}$ ).
- Summand corresponding to accidental actions:  $S_{A_r,v_A}$  ( $1 \leq v_A \leq 2^{n_A}$ ).
- Summand corresponding to seismic actions:  $S_{AS_r,v_{AS}}$  ( $1 \leq v_{AS} \leq 2^{n_{AS}}$ ).

**A.6.4.1.1 Combinations of actions for persistent or transient design situations** With this notation, the expression (A.2) can be written as follows:

$$CQ_{v_G,v_{G*},v_Q,d} = S_{G_k,v_G} + S_{G_{*k},v_{G*}} + S_{Q_{r0,d},v_Q} \quad (\text{A.18})$$

where:

$v_G$  is the variation corresponding to the permanent actions;

$v_{G*}$  is the variation corresponding to the permanent actions of a non-constant value;

$v_Q$  is the variation corresponding to the variable actions;

$d$  is the index that corresponds to the leading variable action, and

$\mathbf{Q}_{r0,d}$  is the vector  $[Q_{r0,1}, Q_{r0,2}, \dots, Q_{r0,d-1}, Q_{k,d}, Q_{r0,d+1}, \dots, Q_{r0,n_Q}]$

**A.6.4.1.2 Combinations of actions for accidental design situations** Similarly, the expression (A.4) can be written as follows:

$$CA_{v_G,v_{G*},v_Q,d,m} = S_{G_k,v_G} + S_{G_{*k},v_{G*}} + S_{Q_{r2,d},v_Q} + A_{k,m} \quad (\text{A.19})$$

where:

$v_G$  is the variation corresponding to the permanent actions;

$v_{G*}$  is the Variation corresponding to the permanent actions of a non-constant value;

$v_Q$  is the variation corresponding to the variable actions;

$d$  is the index corresponding to the leading variable action;

$\mathbf{Q}_{r2,d}$  is the vector  $[Q_{r2,1}, Q_{r2,2}, \dots, Q_{r2,d-1}, Q_{r1,d}, Q_{r2,d+1}, \dots, Q_{r2,n_Q}]$ ;

$m$  is the index that corresponds to the accidental action considered, and

$A_{k,m}$  is the design value of the accidental action  $m$ .

**A.6.4.1.3 Combinations for seismic design situations** Similarly, the expression (A.6) can be written as follows:

$$CS_{v_G, v_{G*}, v_Q, n} = S_{G_k, v_G} + S_{G_{*k}, v_{G*}} + S_{Q_{r2}, v_Q} + AS_{k,n} \quad (\text{A.20})$$

where

$v_G$  is the variation corresponding to the permanent actions;

$v_{G*}$  is the variation corresponding to the permanent actions of a non-constant value;

$v_Q$  is the variation corresponding to the variable actions;

$\mathbf{Q}_{r2}$  is the vector  $[Q_{r2,1}, Q_{r2,2}, \dots, Q_{r2,n_Q}]$ ;

$n$  is the index of the seismic action considered, and

$AS_{k,n}$  is the design value of the seismic action  $n$ .

**A.6.4.1.4 Calculation algorithm** The proposed algorithm for writing all the combinations for ultimate limit states is as follows:

1. calculation of all the variations corresponding to actions G:  $\gamma_{g, v_G}$  ( $1 \leq v_G \leq 2^{n_G}$ )
2. calculation of all the variations corresponding to actions G\*:  $\gamma_{g*, v_{G*}}$  ( $1 \leq v_{G*} \leq 2^{n_{G*}}$ )
3. calculation of all the variations corresponding to actions Q:  $\gamma_{q, v_Q}$  ( $1 \leq v_Q \leq 2^{n_Q}$ )
4. from  $d = 1$  to  $d = n_q$ 
  - (a) calculation of all the combinations  $CQ_{v_G, v_{G*}, v_Q, d}$ .
5. from  $d = 1$  to  $d = n_Q$ 
  - (a) from  $m = 1$  to  $m = n_A$ 
    - i. calculation of all the combinations  $CA_{v_G, v_{G*}, v_Q, d, m}$ .
6. from  $n = 1$  to  $n = n_{AS}$ 
  - (a) calculation of all the combinations  $CS_{v_G, v_{G*}, v_Q, n}$ .
7. end

refinement of step 4a:

1. from  $v_G = 1$  to  $v_G = 2^{n_G}$ 
  - (a) calculate  $S_{G_k, v_G}$
  - (b) from  $v_{G*} = 1$  to  $v_{G*} = 2^{n_{G*}}$ 
    - i. calculate  $S_{G_{*k}, v_{G*}}$



- ii. from  $v_Q = 1$  to  $v_Q = 2^{n_Q}$ 
  - A. calculate  $S_{Q_{r0,d},v_Q}$
  - B. calculate  $CQ_{v_G,v_{G*},v_Q,d} = S_{G_k,v_G} + S_{G_{*k},v_{G*}} + S_{Q_{r0,d},v_Q}$

2. end

refinement of step 5(a)i:

- 1. from  $v_G = 1$  to  $v_G = 2^{n_G}$ 
  - (a) calculate  $S_{G_k,v_G}$
  - (b) from  $v_{G*} = 1$  to  $v_{G*} = 2^{n_{G*}}$ 
    - i. calculate  $S_{G_{*k},v_{G*}}$
    - ii. from  $v_Q = 1$  to  $v_Q = 2^{n_Q}$ 
      - A. calculate  $S_{Q_{r2,d},v_Q}$
      - B. calculate  $CA_{v_G,v_{G*},v_Q,d,m} = S_{G_k,v_G} + S_{G_{*k},v_{G*}} + S_{Q_{r2,d},v_Q} + A_{k,m}$

2. end

refinement of step 6a:

- 1. from  $v_G = 1$  to  $v_G = 2^{n_G}$ 
  - (a) calculate  $S_{G_k,v_G}$
  - (b) from  $v_{G*} = 1$  to  $v_{G*} = 2^{n_{G*}}$ 
    - i. calculate  $S_{G_{*k},v_{G*}}$
    - ii. from  $v_Q = 1$  to  $v_Q = 2^{n_Q}$ 
      - A. calculate  $S_{Q_{r2},v_Q}$
      - B. calculate  $CS_{v_G,v_{G*},v_Q,n} = S_{G_k,v_G} + S_{G_{*k},v_{G*}} + S_{Q_{r2},v_Q} + AS_{k,n}$

2. end

#### A.6.4.2 Combinations for serviceability limit states

Taking into account the partial factors for serviceability limit states, if:

$$S_{G_k} = \sum_{i=1}^{n_G} G_{k,i} \quad (\text{A.21})$$

$$S_{G_{*k}} = \sum_{j=1}^{n_{G*}} G_{*k,j} \quad (\text{A.22})$$

$$S_{Q_{r0,d}} = \sum_{l=1}^{d-1} Q_{r0,l} + Q_{k,d} + \sum_{l=d+1}^{n_Q} Q_{r0,l} \quad (\text{A.23})$$

$$S_{Q_{r2,d}} = \sum_{l=1}^{d-1} Q_{r2,l} + Q_{r1,d} + \sum_{l=d+1}^{n_Q} Q_{r2,l} \quad (\text{A.24})$$

and

$$S_{Q_{r2}} = \sum_{l=1}^{n_Q} Q_{r2,l} \quad (\text{A.25})$$

then: the  $n_Q$  rare combinations will be:

$$CPF_d = S_{G_k} + S_{G^*_{*k}} + S_{Q_{r0,d}} \quad (\text{A.26})$$

the  $n_Q$  frequent combinations will be:

$$CF_d = S_{G_k} + S_{G^*_{*k}} + S_{Q_{r2,d}} \quad (\text{A.27})$$

and the quasi-permanent combination will be:

$$CCP = S_{G_k} + S_{G^*_{*k}} + S_{Q_{r2}} \quad (\text{A.28})$$

**A.6.4.2.1 Calculation algorithm** The calculation algorithm of all the combinations for serviceability limit states would be expressed as follows:

1. calculation of  $S_{G_k}$
2. calculation of  $S_{G^*_{*k}}$
3. from  $d = 1$  to  $d = n_Q$ 
  - (a) calculate  $S_{Q_{r0,d}}$
  - (b) calculate  $CPF_d = S_{G_k} + S_{G^*_{*k}} + S_{Q_{r0,d}}$
4. from  $d = 1$  to  $d = n_Q$ 
  - (a) calculate  $S_{Q_{r2,d}}$
  - (b) calculate  $CF_d = S_{G_k} + S_{G^*_{*k}} + S_{Q_{r2,d}}$
5. calculation of  $S_{Q_{r2}}$
6. calculate  $CCP = S_{G_k} + S_{G^*_{*k}} + S_{Q_{r2}}$
7. end

# Bibliography

- [1] Carlos A. Felippa, *Introduction To Finite Element Methods (ASEN 5007)*. (Department of Aerospace Engineering Sciences University of Colorado at Boulder).
- [2] A. López, D. J. Yong, M. A. Serna, *Lateral-torsional buckling of steel beams: a general expression for the moment gradient factor*. (Lisbon, Portugal: Stability and ductility of steel structures, 2006).
- [3] Ministerio de Fomento, *EHE; Instrucción de hormigón estructural*. (España: Comisión Permanente del Hormigón.Ministerio de Fomento. 1998).
- [4] Ministerio de Fomento, *EAE; Instrucción de acero estructural*. (España: Comisión Permanente de estructuras de acero.Ministerio de Fomento. 2004).
- [5] Ministerio de Fomento, *IAP; Instrucción sobre las acciones a considerar en el proyecto de puentes de carretera*. (España: Dirección General de Carreteras. Ministerio de Fomento. 1998).
- [6] Ministerio de Fomento, *NCSE-02; Norma de construcción sismorresistente: parte general y edificación*. (España: Comisión permanente de Normas Sismorresistentes. Ministerio de Fomento. 2002).
- [7] Ministerio de Fomento, *NBE-AE-88; Acciones en la edificación*. (España: Ministerio de Fomento. 1988).