Tutorial 2. Analysis of a vibrating string under tension.

This tutorial deals with a flexible string fixed at the ends and stretched to an initial strain. The goal is to find its first three natural frequencies of lateral vibration.

In the example, a uniform steel string of length $L = 2m$ and cross-sectional area $A = 2.0e-6m^2$, is fixed at the ends and stretched to an initial strain $\epsilon_0 = 0.005$. The Young's modulus of steel is $E_s = 210\ GPa$ and its specific mass is $\rho_s = 7850kg/m^3$.

Import modules. Firstly, we execute some `import` statements, so that the code in our script gains access to the code in the imported modules.

```
1 from __future__ import division
2 import math
3 import xc_base
4 import geom
5 import xc
6 from model import predefined_spaces
7 from materials import typical_materials
```

Listing 1: Imported modules.

The `future division` statement (line 1) change, throughout the module, the / operator to mean true division instead of returning the floor of the mathematical result of division when the arguments are `ints` or `longs` (transitional measure from Python 2.x to the standard meaning in Python 3.0).

The `math` module (line 2) provides access to several mathematical functions (`floor, exp, sqrt, cos, sin` ...).

The following three lines correspond with three main modules of XC :

- `xc_base`: includes the basic functions for the Python interface: assign and retrieve properties stored in the C++ classes (see example `test_evalPy.py`) and execute Python scripts (see example `test_execPy.py`).
- `geom`: handles entities related to geometry, like points, lines, polylines, planes, polygons, circles, coordinate systems, grids, vectors, matrices, rotations, translations, ...
- `xc`: this module provides access to the finite element classes and functions: mesh generation, element type and material definition, analysis, ...

Import statements in lines 6-7 have to do with the following modules:

- `predefined_spaces`: this module is intended to set the dimension of the space and the number of nodal DOF, as well as to introduce constraints to them.
- `typical_materials`: several often-used materials are predefined in this module: elastic uniaxial with or without tension branch, prestressed cable, concrete, steel, ...

Definition of parameters. XC allows full parametric models, that's to say, the definition of geometry, material, loads, ..., can be based on properties that, if your data change, the problem is recalculated accordingly.

Lines 8 to 14 set the value of the parameters which will be used later during the model generation.

```
8  E= 2.1e11 # steel Young modulus [Pa]
9  ro=7850   #steel specific mass [kg/m3]
10 l= 2.0 # String length [m]
11 sigmaPret= E*0.005 # Prestressing stress [Pa]
12 area= 2.0e-6      # Cross-sectional area [m2]
13 Mass= ro*area # Mass per unit length [kg/m]
14 NumDiv= 13    #number of elements
```

Listing 2: Parameters.

Finite element problem. The type of problem defined is StructuralMechanics2D (line 18), that's to say, nodes are defined by two coordinates (x,y) and has three degrees of freedom (u_x, u_y, θ). This function takes as argument the handler of nodes, that is retrieved from preprocessor in line 17.

```
15 feProblem= xc.FEProblem()
16 preprocessor=  feProblem.getPreprocessor
17 nodes= preprocessor.getNodeHandler
18 modelSpace= predefined_spaces.StructuralMechanics2D(nodes)
```

Listing 3: FE problem type.

Definition of material The material defined in line 19 is an elastic uniaxial material that can not withstand compression. Its strain-stress relationship ranges from slack (large strain at zero stress) to taught (linear with modulus E). The method takes as parameters the preprocessor, a name that we'll use to assign this material to the elements, its Young's modulus, the prestressing force and the mass per unit length.

```
19 typical_materials.defCableMaterial(preprocessor, "cable",E,
       sigmaPret,Mass)
```

Listing 4: Materials.

→ Find out more about materials in XC

Definition of the element type. The type of element used lately to mesh the model is defined in lines 20 to 25. It is a two-dimensional co-rotational truss, allowed to have large displacements and rotations at the global level by means of the co-rotational formulation. This formulation seeks to separate rigid body motions (rotation and translation) from strain producing deformations at the local element level. A co-rotating frame is attached to the truss with origin at its first node and x-axis directed along the element. This local coordinate system translates and rotates with the truss, so that with respect to it only local strain producing deformations along the x-axis remains.

```
20 seedElemHandler= preprocessor.getElementHandler.
       seedElemHandler
21 seedElemHandler.defaultMaterial= "cable"
22 seedElemHandler.dimElem= 2 # Dimension of element space
23 seedElemHandler.defaultTag= 1 #Tag for the next element.
24 truss= seedElemHandler.newElement("CorotTruss",xc.ID([0,0]))
25 truss.area= area
```

Listing 5: Element type.

→ Find out more about element types in XC

Definition of the points and lines. Two points and the line joining them is created in lines of code 26 to 31.

```
26 points= preprocessor.getMultiBlockTopology.getPoints
27 pt1= points.newPntIDPos3d(1,geom.Pos3d(0.0,0.0,0.0))
28 pt2= points.newPntIDPos3d(2,geom.Pos3d(1,0.0,0.0))
29 lines= preprocessor.getMultiBlockTopology.getLines
30 lines.defaultTag= 1
31 l= lines.newLine(pt1.tag,pt2.tag)
```

Listing 6: Points and lines.

Meshing The line created is meshed using the type of element defined in listing 5 and set as default. The number of elements created is given by the number of divisions in the line, defined by means of its attribute `nDiv`.

```
33 l.nDiv= NumDiv
34 l.genMesh(xc.meshDir.I)
```

Listing 7: Mesh.

Definition of constraints. Lines 34 and 35 introduce single-point boundary constraints in the extreme nodes of the line, restraining all the three degrees of freedom (000 → $u_x = 0, u_y = 0, \theta = 0$) in the start node and only rotation (FF0 → $u_x = free, u_y = free, \theta = 0$) in the end node.

```
34 predefined_spaces.ConstraintsForLineExtremeNodes(l,modelSpace.
       fixNode000)
35 predefined_spaces.ConstraintsForLineInteriorNodes(l,modelSpace
       .fixNodeFF0)
```

Listing 8: Constraints.

→ Find out more about boundary conditions in XC

Obtaining static solution. In lines 36 to 57 the static analysis is defined and performed. XC is provided with some pre-defined solvers, available in module `typical_analysis`. In this example, all the components that integrate the analysis are detailed one by one. The first four lines refer to the creation of the model-wrapper and the analysis-strategy containers. The first one is a container

of domain, analysis, constraint handler and DOF-numberer, and has to do with the finite element model "through the eyes" of the solver. The second one is a container to which aggregate the model wrapper and other components of the analysis definition.

In line 42 a model wrapper is created for the new analysis. Then, a constraint handler of type `plain_handler` is added to it. That handler determines how the constraint equations are enforced in the analysis, and may be able of managing single-point constraints (g.e. support conditions) or multi-point constraints. The use of multi-point constraints can arise in a FE model, g.e., to enforce an equal-displacement behavior (i.e. translations and/or rotations at different nodes are constrained to be equal), to impose a rigid body behavior (i.e., displacements at different nodes are related as it connected by rigid links), or to reproduce symmetry and antisymmetry conditions. The plain-constraint handler used in the example can enforce homogeneous single point constraints and multi-point constructed where the constraint matrix is equal to the identity.

Next, the DOF numberer is added to the model wrapper. That object determines how the degrees of freedom are numbered. As the problem is small, a plain numberer (`simple`) is chosen, which means that the numberer takes the nodes whatever the order the domain gives them and numbers those nodes consequently.

```
36 solProc=feProblem.getSoluProc
37 solCtrl= solProc.getSoluControl
38 solModels= solCtrl.getModelWrapperContainer
39 analAggrContainer= solCtrl.getSolutionStrategyContainer
40 #static analysis
41 sm= solModels.newModelWrapper("sm")
42 cHandler= sm.newConstraintHandler("plain_handler")
43 numberer= sm.newNumberer("default_numberer")
44 numberer.useAlgorithm("simple")
45 solutionStrategy= analAggrContainer.newSolutionStrategy("
       solutionStrategy","sm")
46 solAlgo= solutionStrategy.newSolutionAlgorithm("
       newton_raphson_soln_algo")
47 ctest= solutionStrategy.newConvergenceTest("
       norm_unbalance_conv_test")
48 ctest.tol= 1e-8
49 ctest.maxNumIter= 100
50 integ= solutionStrategy.newIntegrator("load_control_integrator
       ",xc.Vector([]))
51 Nstep= 10  #  apply load in 10 steps
52 DInc= 1./Nstep  #  first load increment
53 integ.dLambda1= DInc
54 soe= solutionStrategy.newSystemOfEqn("band_gen_lin_soe")
55 solver= soe.newSolver("band_gen_lin_lapack_solver")
56 analysis= solProc.newAnalysis("static_analysis","
       solutionStrategy","")
57 result= analysis.analyze(Nstep)
```

Listing 9: Static solution.

In line of code number 46, the solution algorithm is defined. That algorithm determines the sequence of steps taken to solve the non-linear equation; for this case the well-known Newton-Raphson method is applied, i.e. the set of nonlinear equations is locally linearized and solved. If the solution does not satisfy the original nonlinear equations, the latter are again linearized at the new solution.

A convergence criterion is defined in line 47 to determine if convergence has been achieved at the end of an iteration step. The norm unbalance test uses the norm of the right hand side of the matrix equation for comparison. The tolerance adopted is `1e-8` and the maximum number of iterations to check before failure condition is taken equal to 100.

Next, an integrator object is defined (lines 50 to 53), which will be involved in the numerical integration required by the computation of the stiffness matrix and load vectors. The integrator is used to determine the predictive step for time t+dt (in the example the set value of load factor increment is 1/10), to specify the tangent matrix and residual vector at any iteration, and to determine the corrective step based on the displacement increment.

Lines 54 and 55 define the algorithm used to solve the global equations. The method chosen corresponds to a matrix system that has a banded profile. When a solution is required, the Lapack (**L**inear **A**lgebra **Pack**age) routines are used.

Finally a static analysis is performed in ten steps.

→ Find out more about analysis in XC

Review results of static analysis. We ask one element of the string its stress and axial internal force.

```
57 elements= preprocessor.getElementHandler
58 ele1= elements.getElement(1)
59 tension= ele1.getN()
60 sigma= ele1.getMaterial().getStress()
61 print "stress= ",sigma
62 print "tension= ",tension
```

Listing 10: Results static analysis.

The result must be:

$$\sigma = E_s \times \epsilon_0 = 1050e6Pa$$

$$F = \sigma \times A_{cross\ section} = 2100.0N$$

Obtaining eigensolution. A modal analysis is performed in order to get the first three eigenvalues. The mapping between the degrees of freedom and the equation numbers is performed by using an `rcm` numberer (reverse Cuthill-McKee algorithm). To solve the problem an algorithm `frequency_soln_algo`, appropriate for solving standard eigenvalue equations, is used.

```
63 sm= solModels.newModelWrapper("sm")
64 cHandler= sm.newConstraintHandler("
       transformation_constraint_handler")
65 numberer= sm.newNumberer("default_numberer")
66 numberer.useAlgorithm("rcm")
67 solutionStrategy= analAggrContainer.newSolutionStrategy("
       solutionStrategy","sm")
68 solAlgo= solutionStrategy.newSolutionAlgorithm("
       frequency_soln_algo")
69 integ= solutionStrategy.newIntegrator("eigen_integrator",xc.
       Vector([]))
70 soe= solutionStrategy.newSystemOfEqn("sym_band_eigen_soe")
71 solver= soe.newSolver("sym_band_eigen_solver")
```

```
analysis= solProc.newAnalysis("eigen_analysis","
    solutionStrategy","")
Neigen=3
analOk= analysis.analyze(Neigen)
```

Listing 11: Eigensolution.

→ Find out more about analysis in XC

Review results of eigenanalysis. Finally, we check the results outcome from the eigenvalue calculation.

```
eig1= analysis.getEigenvalue(1)
eig2= analysis.getEigenvalue(2)
eig3= analysis.getEigenvalue(3)
f1= math.sqrt(eig1)/(2*math.pi)
f2= math.sqrt(eig2)/(2*math.pi)
f3= math.sqrt(eig3)/(2*math.pi)
print "eig1= ",eig1
print "eig2= ",eig2
print "eig3= ",eig3
print "f1= ",math.sqrt(eig1)/(2*math.pi)
print "f2= ",math.sqrt(eig2)/(2*math.pi)
print "f3= ",math.sqrt(eig3)/(2*math.pi)
```

Listing 12: Results eigenanalysis.

The natural frequencies for a string under conditions such as those in this example, can be obtained by:

$$f_n = \frac{n}{2L}\sqrt{\frac{T}{\rho}}, \quad n = 1, 2, 3, \cdots$$

By applying this equation for n= 1,2,3 with $\sqrt{\frac{T}{\rho_s}} = 365.7\ m/s$, we obtain the exact value of the first three frequencies:

$$f_1 = \frac{1}{2L}c = 91.432Hz$$
$$f_2 = \frac{2}{2L}c = 182.865Hz$$
$$f_3 = \frac{3}{2L}c = 274.297Hz$$

that must coincide, approximately, with the three values obtained from the eigenanalysis.