Solidspy_DYN: A Python based finite element code for dynamic analysis of elastic solids using implicit time integration

Juan Gomez and Nicolas Guarin

Abstract

This tutorial describes the basic steps required in the construction of a finite element model using the Python based code **Solidspy_DYN**. The code has been written mainly for teaching purposes and also as a first hand calculator for complex wave propagation problems. **Solidspy_DYN** is written using Python but also makes extensive use of free open code software as **gmsh** and **Paraview** which are used for pre and post processing. This document describes the steps involve in the creation of a **gmsh** model to be used within the implicit time integrator. This is achieve through a sample problem that is geometrically simple yet it involves all the basic parts that conform a **Solidspy_DYN** analysis. It is assumed that the user is familiar with general **gmsh** syntaxis and features. The user manual is focused in the aspects required to complete a dynamic analysis thus emphasizing the visualization of the solution in the free code **Paraview**.

1 Introduction

The finite element method (FEM) or its variations, is probably the numerical technique of major use in modern engineering to the extent that it is included in a large number of engineering curriculum at the undergraduate and graduate level. Unfortunately many of these courses use as simulation tool specialized and expensive commercial software that, although powerful from a computational point of view is highly limited when used for teaching purposes as these codes are usually a black-box hiding most of its numerical methods to the student.The python based open code **SolidsPy** has been written to be used mainly as a teaching tool since it is structured in such a way that its independent subroutines can be used to show the student the different steps in the construction of a finite element algorithm. Furthermore, the code is open and can be modified to satisfy the needs of

a learning process without any risk as it can be restored to its original version.

This document describes the steps required for the generation of a finite element model for elastic two-dimensional media to be analyzed with **Solidspy_DYN**. It walks the student/user through the complete modeling process starting with the definition of the domain using external free open software, continuing with the writing of input files for the code and finishing with solution visualization with additional external free open software.

The first part of the document describes how to define input data files to **Solidspy_DYN** in terms of simple text files. The next section shows the construction of complex geometric domains using **gmsh**, the external and free mesh generation program. Then the manual describes the process of writing the input files using yet another external software corresponding in this case to the python based module **meshio**. The final part of the document describes the postprocessing or visualization with **Paraview**.

In order to show the modeling process we use a simple problem and its **gmsh** definition in terms of geometrical and physical entities as required to conduct dynamic analysis with **Solidspy_DYN**. The model to be analyzed is geometrically simple but it involves all the parts and steps required for the analysis. One of the main purposes in a dynamic analysis of continuum media is to capture the time history of the response of a given quantity. By default **Solidspy_DYN** writes the response time history in terms of displacements at every time step. The solution is written in the form of **vtk** files ready to b visualized with **Paraview**. This tutorial assumes that the user is familiar with the fundamental aspects of **gmsh** and thus it is more focused in the parts required in the model so it can be analyzed with **Solidspy_DYN**.

To be able to follow this tutorial the user requires the following software:

- **Solidspy_DYN**: The dynamic version of the finite element analysis code **Solidspy**.
- **Solidspy**: This basic code is required since we will used some of the libraries available in **Solidspy**.
- **Gmsh**: The free mesh creation software.
- **meshio**: The input-output module to read and write the **Gmsh** models.
- **paraview**: The free solutions visualization software.

2 Creating a model in SolidsPy_DYN

In general, a finite element model for the solution of the theory of elasticity boundary value problem (BVP) involves (i) a discretization or partition of the problem domain into sub-domains, also known as the mesh, (ii) a description of the boundary conditions and (iii) a description of the material properties. Dynamic problems also require the definition of time parameters like initial conditions and a description o the time behavior of loads.

A finite element mesh is nothing else than a discrete set of points (or nodes) arranged into so-called finite elements or subdomains. The method itself, is based upon the local solution of the BVP in each element within which the displacement field at an arbitrary point is expressed in terms of the displacements at the nodal points. Once the displacement field is known it can be used in the determination of strains and stresses. The boundary conditions correspond to prescribed values of the displacements or loads at a given set of nodal points. Similarly, the material properties are just the values of the material parameters associated to the different materials present in the model.

In **SolidsPy_DYN** a finite element model is defined in terms of text files containing the nodal data (nodes.txt), the element data (eles.txt), the loading data (loads.txt), the material data (mater.txt) and in dynamic problems the time integration data (inipar.txt). The data structure of these text files will be described later.

There are several issues in the construction of a finite element model. First, if the domain has a complex geometry its partition into sub-domains becomes difficult. Second, if the corresponding mesh is moderately fine the number of nodes and elements results into large nodal and elemental text files. Here the problem of discretizing the arbitrary complex geometry into finite elements is solved using the free and open 3D finite element mesh generator **Gmsh**. Once the geometry is discretized with **Gmsh** the resulting files need to be converted into **SolidsPy_DYN** text files. This is achieved using **meshio**[1], an input-output python based module for various mesh formats, including **Gmsh**. The complete model creation process is schematized in fig. 1. The generation of the **SolidsPy_DYN** text files using meshio is completed using a problem dependent Python script written using subrotuines available in the preprocesor module in **SolidsPy_DYN**. The generation of the preprocessing script is described in section 4 .

[1] `<https://github.com/nschloe/meshio>`

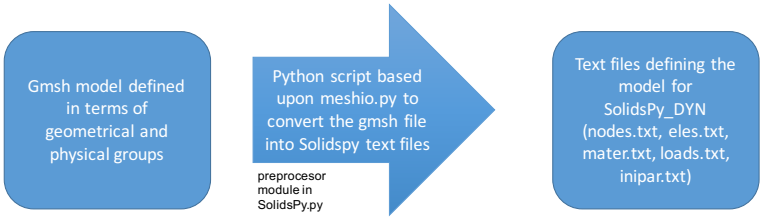

Figure 1: Creation of a SolidsPy model. The geometry is built and discretized with Gmsh and SolidsPy text files are later written out by meshio.py.

In what follows we make a description of each one of the 5 **SolidsPy_DYN** text files.

2.1 Elements data

Each **SolidsPy_DYN** finite element model requires a text file (eles.txt) with the following information for each element in the mesh:

```
ele_id   ele_tag  mat_tag   n1   n2.....nN
```

where

- **ele_id**: Integer defining the id for the current element. The elements in the text file must be listed in consecutive ascending order.
- **ele_tag**: Integer defining the type of element according to (1): 4-noded bi-lineal element, (2): 6-noded quadratic triangle, (3): 3-noded linear triangle.
- **mat_tag** : An integer number defining the material profile associated to the current element. A material profile is a st of material parameters defining the elastic properties of a given material. These are defined in the mater.txt file.
- **n1 n2 ...nN**: For a N-noded element a list of N nodal numbers defining the nodes of the current element.

The eles.txt file can be created out of a Gmsh file by the python subroutine **ele_writer()** available in the **preprocesor** module of the program.

2.2 Nodal data

Each **SolidsPy_DYN** finite element model requires a text file (nodes.txt) with the following information for each nodal point in the mesh

```
id    x-coord    y-coord  BC-x  BC-y
```

where

- **id**: Integer defining the id for the current node. The nodes in the text file must be listed in consecutive ascending order.
- **x-coord**: Horizontal coordinate of the nodal point in a cartesian reference system.
- **y-coord**: Vertical coordinate of the nodal point in a cartesian reference system.
- **BC-x**: Horizontal displacement boundary condition flag at the current node. A value of -1 implies that the degree of freedom is restrained and a value of 0 that it is free.
- **BC-y**: Vertical displacement boundary condition flag at the current node. A value of -1 implies that the degree of freedom is restrained and a value of 0 that it is free.

The nodes.txt file can be created out of a Gmsh file by the python subroutine **node_writer()** available in the **preprocesor** module of the program. Since the displacement boundary conditions are different than 0 only at a limited set of nodes these can be written using the python subroutine **boundary_conditions()** available in the **preprocesor** module of the program.

2.3 Materials data

Each **SolidsPy_DYN** finite element model requires a text file (mater.txt) with the following information for each material profile in the mesh. A material profile is a set of parameters corresponding to a given material. Each material profile is defined by a line of text with the following parameters:

```
E    nu     rho   beta  alpha
```

where

- E : Young modulus
- ν : Poisson's ratio.
- ρ : Mass density.
- D_1 : Constant for mass proportional damping.
- D_2 :Constant for stiffness proportional damping.

The mater.txt file must be manually created by the user.

2.4 Loading data

Each **SolidsPy_DYN** finite element model requires a text file (loads.txt) with the following information for each loaded node in the mesh.

```
Node Fx   Fy
```

where

- **Node**: Node number for the current load.
- F_x: Magnitude of the nodal point load in the horizontal direction.
- F_y: Magnitude of the nodal point load in the vertical direction.

SolidsPy_DYN only considers nodal point loads, and thus surface tractions must be converted into nodal point loads. This conversion can be made using the Gmsh file and the python subroutine **loading()** available in the **preprocesor** module of the program.

2.4.1 Time dependence of the loading data

All the loads in **Solidspy_DYN** are specified with a time variation in terms of a Ricker pulse. This time function is defined as follows:

$$A = \left(1 - 2\pi^2 f_c^2 t^2\right) e^{-\pi^2 f_c^2 t^2}$$

where f_c is the central frequency. In **Solidspy_DYN** the pulse is centered as specified by t_c.

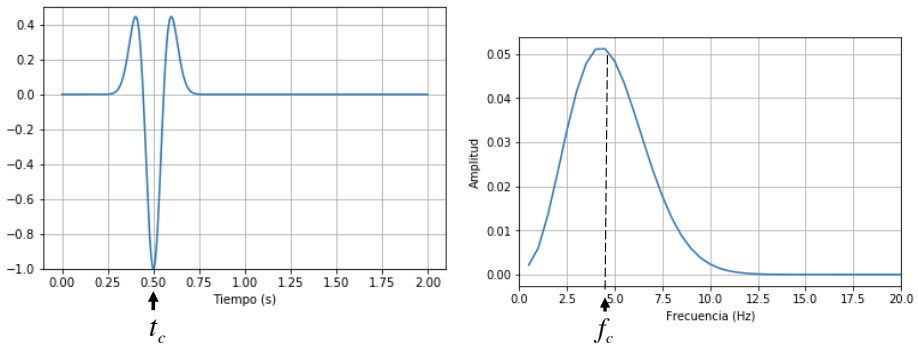

Figure 2: Time function and Fourier spectra for the Ricker pulse. The signal is characterized by its central time t_c and its dominant frequency parameter f_c

The load is defined using data combined in two files. In the loads.txt file the user must specify a list of nodes with the amplitude of the load

along each degree of freedom. The remaining parameters corresponding to the time variation of the load and completely defining the Ricker pulse are indicated in the integration parameters file of the model as described next.

2.5 Time dependent data

Each **SolidsPy_DYN** finite element model requires a text file (inipar.txt) with the following time dependent information for the analysis:

```
dt T  tc fc
```

where

- dt : Time step for the incremental analysis.
- T : Duration of the analysis.
- t_c : Central time in the Ricker pulse.
- f_c : Central frequency in the Ricker pulse.

3 Definition of a finite element model for SolidsPy_DYN using Gmsh

This section describes the basic components and steps for the construction of a finite element model to conduct dynamic analysis with **Solidspy_DYN**. It is assumed that the user is familiar with Gmsh. There are several tutorials in the TUTORIALS folder of the code.

The model construction process is explained with reference to the problem shown in fig. 3 corresponds to a rectangular plate composed of two different materials and submitted to a point load with time variation defined by a Ricker pulse applied at the top.

The construction of a model in **Gmsh** involves the generation of geometrical elements defining all the points, lines and planes required to build the geometry and physical groups corresponding to special sets of points, lines and planes (or surfaces) where particular modeling conditions are expected to occur. The names or labels assigned to the physical groups are later associated to the specific elements of the mesh pertaining to these groups.

For instance if nodal points are to be applied over a given line of the model, it is convenient to define that specific line as physical line. Later during the mesh creation process all the nodes lying along that specific physical line will be stored in an independent data structure facilitating the modeling process. The following sections describe the

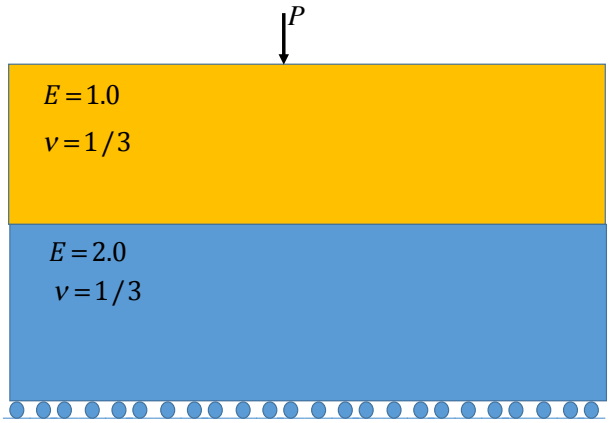

Figure 3: Rectangular plate conformed by two different materials under a time varying point load at the free surface.

generation of the geometrical and physical groups required in a **Gmsh** model for **SolidsPy_DYN**[2].

3.1 Geometrical entities

The geometrical entities of the model are shown in fig. 4 together with the conventions used to describe each entity. The corresponding entities or groups are in hierarchical order points, lines (formed by points), line loops (formed by groups of lines generating a closed figure) and plane surfaces (formed by a line loop). The current model contains two plane surfaces since we will be using two different materials in each part of the domain. When creating the model it is convenient to use parameters for all the dimensions as this will facilitate the execution of several analysis later. Here we will assume that the model is of width L, while the lateral faces are specified by dimensions $H1$ and $H2$. As additional parameter we will use the characteristic element size hc to be specified as a fraction of the total width.

3.2 Physical entities

The physical groups in the model are described in fig. 5. These correspond to the two physical surfaces labeled (100) and (200) respectively; a physical line, labeled (400) and a physical point with the label (500). These names will be assigned to all the nodes and elements that coincide with those specific groups. The physical surfaces are required

[2]The following link contains a series of videos showing the modeling process for this example `<https://www.youtube.com/channel/UCNhX9Z5wkEk_JhlSuIo8A4Q>`

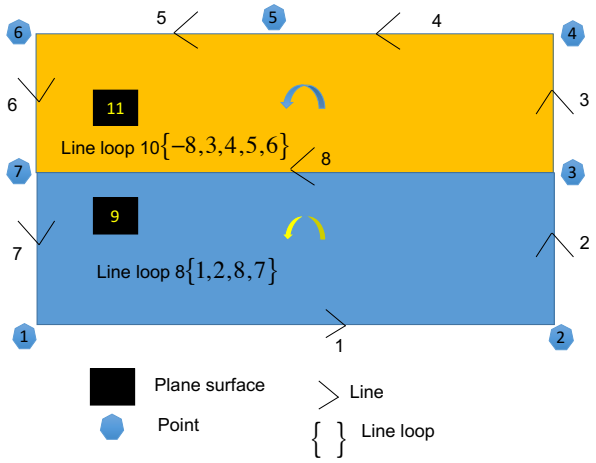

Figure 4: Geometrical entities for the model.

in order to indicate which elements are assigned each material profile. The physical line is required in order to identify the nodes at which displacement boundary conditions are to be imposed and the physical node is required in order to specify the loading node. Each physical group is described next:

- Physical surface 100: Top material layer.
- Physical surface 200: Bottom material layer.
- Physical line 400: Bottom surface with all restrained vertical and horizontal displacements.
- Physical point 500: Localization of the dynamic point load.

The model defined in terms of geometrical and physical groups is returned by as a file with extension `.geo`. The specific `.geo` file for the example problem is shown next.

```
// Parameters
L = 6.0;
H1 = 3.0;
H2 = 3.0;
hc = L/10.0;
// Points
Point(1) = {0.0, 0.0, 0, hc};
Point(2) = {L, 0.0, 0, hc};
Point(3) = {L, H1, 0, hc};
Point(4) = {L, H1+H2, 0, hc};
```

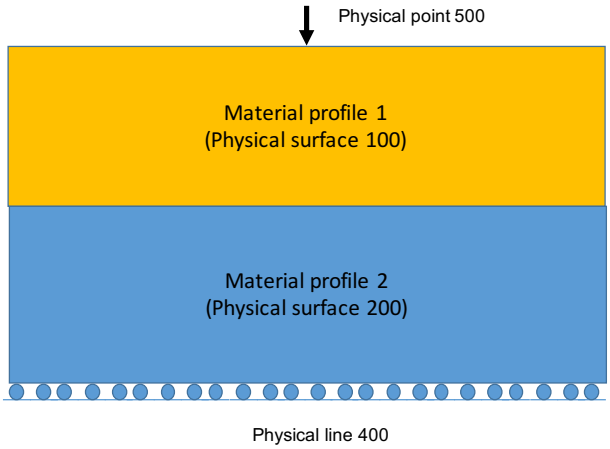

Figure 5: Physical entities for the model.

```
Point(5) = {L/2, H1+H2, 0, hc};
Point(6) = {0, H1+H2, 0, hc};
Point(7) = {0, H1, 0, hc};

// Lines
Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 5};
Line(5) = {5, 6};
Line(6) = {6, 7};
Line(7) = {7, 1};
Line(8) = {3, 7};

// Surfaces
Line Loop(8) = {1, 2, 8, 7};
Plane Surface(9) = {8};
Line Loop(10) = {-8, 3, 4, 5 , 6};
Plane Surface(11) = {10};

// Physical groups
Physical Surface(100) = {9};  // Top layer
Physical Surface(200) = {11};  // Bottom layr
Physical Line(400) = {1};  // Bottom line
Physical Point(500) = {5};  // Load point
```

After conducting a couple of mesh refinements the discretized model should look as shown in fig. 6

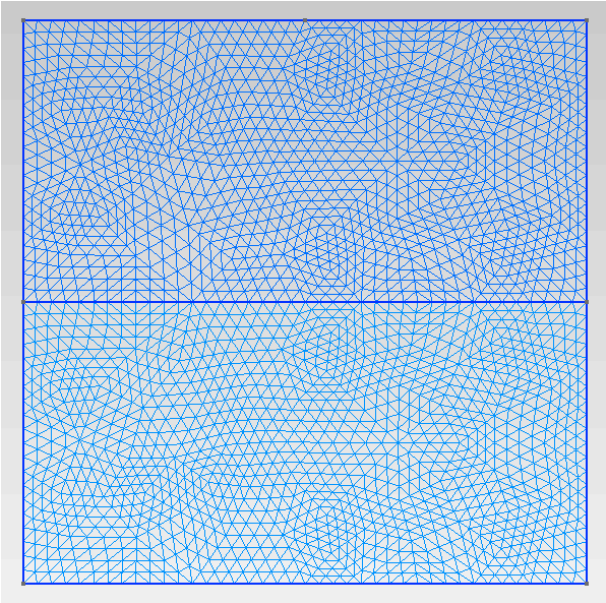

Figure 6: Finite element mesh conformed of linear triangular elements.

4 Generation of text files from the Gmsh model

The input data in terms of text files for the finite element code must now be defined using the **gmsh** model. This is achieved using subroutines available in the **preprocessor** module of **SolidsPy**. These routines are written on top of the input-output python based module **meshio**. These subroutines are essentially required to write the nodes.txt and elements.txt files required by **Solidspy_DYN** which may result massive and cumbersome to process. The python script that produces the input files for the current example is described next.

```
# -*- coding: utf-8 -*-
"""
"""
from __future__ import division, print_function
import meshio
import numpy as np
```

```
import solidspy.preprocesor as msh
#
points, cells, point_data, cell_data, field_data = \
    meshio.read("bimaterial.msh")
#
nodes_array    = msh.node_writer(points , point_data)
nf , els1_array = msh.ele_writer(cells , cell_data , \
                  "triangle" , 100 , 3 , 0 , 0)
nini = nf
nf , els2_array = msh.ele_writer(cells , cell_data , \
                  "triangle" , 200 , 3 , 1 , nini)
#
nodes_array = msh.boundary_conditions(cells , cell_data ,\
                  400 , nodes_array , -1 , -1)
els_array = np.append(els1_array, els2_array , axis = 0)
#
np.savetxt("eles.txt" , els_array   , fmt="%d")
np.savetxt("nodes.txt", nodes_array , fmt=("%d", "%.4f",\
           "%.4f", "%d", "%d"))
```

In the first block we import the required modules for the script. In particular the script requires the module **meshio** and the module **preprocesor** from **solidspy**.

After the modules have been imported, the first line of code uses **meshio** to read the **gmsh** mesh. In this case the mesh has been saved with the name **bimaterial.msh**. Once the mesh is read all the model is stored in python dictionaries ready to be processed. The process of manipulating the dictionaries and extracting the data for the text files is done with subroutines available in the module **preprocesor** from **solidspy** and this implies that for these routines to be available **solidspy** has been previously installed.

In the first part the points and point_data dictionaries are used by the subroutine node_writer to write an array with the nodal point data in the format required by the finite element code.

The element processing is performed by the element writer subroutine. This subroutine must be invoked as many times as there are physical surfaces in the model. The subroutine uses the following input parameters:

- The python dictionaries cells and cell_data storing all the element information.
- The name for the element used in the current surface according to **gmsh** which in this case corresponds to "triangle" for the lineal triangles.

- The number of the surface being processed according to **gmsh**. In this case we have defined physical surfaces with numbers 100 and 200 for the top and bottom surface respectively.
- A number identifying the type of element according to **Solidspy_DYN** which in this case is 3 since we are using a triangular lineal element.
- A number identifying the material profile to be used in the current surface. In this case the top surface has been assigned the material properties defined in the material profile 0 while the bottom surface has been assigned those from material profile 1.
- An integer parameter specifying where the current surface should start counting elements. In this particular problem the element writer starts counting elements from 0 in the first surface and from **nini** in the second surface where **nini** is the last element from the previous surface.

Upon execution this subroutine returns an array with the elemental information for the current surface but now written in **Solidspy_DYN** format and an integer number indicating the total number of elements contained in that surface. This number is required if processing several surfaces since this marks down where the next element counting should begin.

The next part of the script specifies the boundary conditions. Thus boundary conditions subroutine requires the following input parameters:

- The python dictionaries cells and cell_data storing all the element information.
- An integer specifying the name assigned to the line where the current boundary condition is to be imposed. In this case we have used the number 400 to specify the bottom line of the model.
- The array containing the nodal data and generated by the nodes writer subroutine. This array will be modified by the boundary conditions subroutine.
- Two boundary condition specifiers indicating if the horizontal and vertical degrees of freedom are free (value = 0) or restrained (value = -1).

The final part of the script uses the previously generated arrays and writes the corresponding text files. Since we have processed two physical surfaces the two resulting arrays must be concatenated into a single element array ready for output. This is done using the python function append.

5 Visualization of displacement time histories

Animated view of the propagation patterns

In wave propagation analysis it is common to develop conceptual understanding of a given solution based upon the propagation pattern experienced by the propagated wave field. In **Solidspy_DYN** these visualizations can be realized using the external open-free visualization software **Paraview**. By default **Solidspy_DYN** writes ready-to-process VTK files that can be read by **Paraview**. Each output VTK file corresponds to a snapshot of the solution at given time instant. These files are stored by **Solidspy_DYN** in the folder /MESHES/VTKs.

View of the surface displacements at a line of points

A second possibility for the visualization of results is to simultaneously plot the time response for a series of points located along a given line. If the resulting time signals are plotted with the amplitudes progressively scaled by a constant factor the plot gives a physical sense of the propagation pattern.

The generation of results for this type of visualization requires the creation of an additional file, **salida.txt**, storing the identification numbers for the set of response points. This file is created directly from the same script that generates the text files for the model after adding the function **respuesta ()**. This function extracts the nodal identifiers located along a previously defined physical line and writes the required text file.

The second step required to obtain the sheet-like visualization of the response is to execute the analysis within a particular script. The script executes the following steps:

- Executes the code and computes the time history of displacements at all the nodes. This solution is stored in the array **U**.
- Loads the text file **salida.txt** into the array **salida**.
- Invokes the post-processing function **sheets** using as input parameters the array **salida** and the solution array **U**. This function writes down the response into a second file named **respuesta.txt**.
- Load the file **respuesta.txt** into the array **vals**.

- Use either of/or both functions **plot**$_{s}$*heetand***plot**$_{\mathbf{p}}$**color***usingasinputparameterthearray***vals***toplotthesol*

An example of the sheet-like visualization of the results is shown in fig. 7, while the specific function **respuesta ()** that generates the file for output and the script for execution of the code in post-processing mode are included at the end.

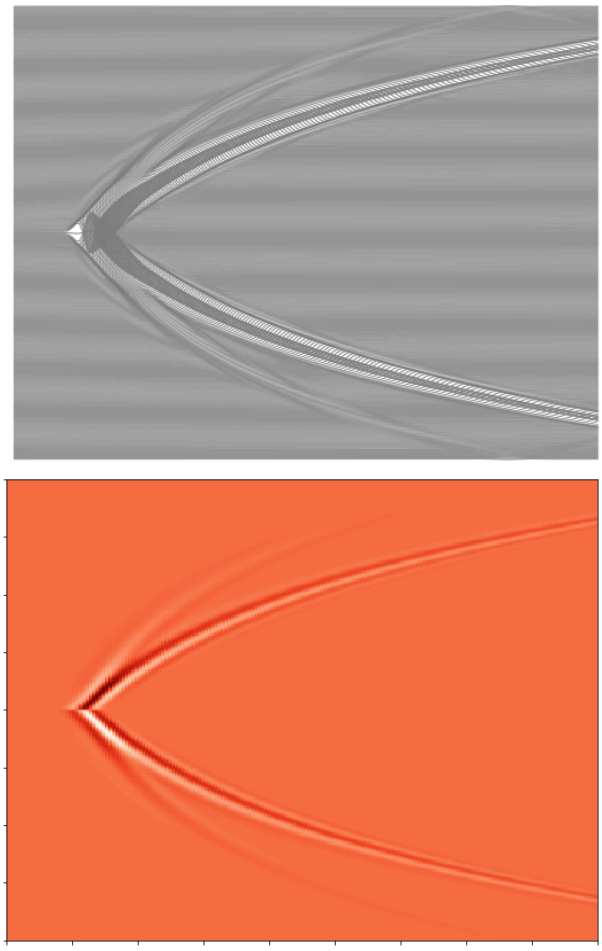

Figure 7: Visualization of response time histories at a set of nodal points located along the same line.

```
def respuesta(cells, cell_data, phy_lin):
    """Extracts the nodes located at the physical line
       phy_line

    Parameters
    ----------
        cell : dictionary
            Dictionary created by meshio with cells information.
        cell_data: dictionary
            Dictionary created by meshio with cells data information.
```

```
        phy_lin : int
            Physical line to print nodal histories.

    Returns
    -------
        nodes_carga : int
            Array with the nodal data corresponding to the physical
            line phy_line.

    """
    lines = cells["line"]
    phy_line = cell_data["line"]["physical"]
    id_carga = [cont for cont in range(len(phy_line))
                if phy_line[cont] == phy_lin]
    nodes_carga = lines[id_carga]
    nodes_carga = nodes_carga.flatten()
    nodes_carga = list(set(nodes_carga))
    nodes_carga.sort(reverse=False)

    return nodes_carga
#
```

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Script para realización de un análisis con post-procesamiento.
Escribe archivos VTKs, regresa el arreglo con la histora de desplazamintos
y grafica historias (en forma de sabanas) para los puntos de la suprficie
definidos en el archivo de texto salida.txt
"""

from __future__ import division, print_function
import numpy as np
import matplotlib.pyplot as plt
from solidsPy_DYN import solidsPy_DYN
import postprocesor as pos
#
U , folder , IBC , ninc , T = solidsPy_DYN()
salida    = np.loadtxt(folder + 'salida.txt' , ndmin = 1 , dtype=np.int)
npts = pos.sheets(salida , ninc , U , IBC , 'respuesta' , folder )
#
vals = np.loadtxt(folder + "respuesta.txt")
plt.close("all")
#
```

```
amp_signal=100
amp_shift =75
plt.figure(0)
fig = plt.figure(figsize=(10, 8))
pos.plot_sheet(vals, T, amp_signal, amp_shift)
#
plt.figure(1)
fig = plt.figure(figsize=(10, 8))
pos.plot_pcolor(vals, T , -1, 1 )
```

References