

# Examples Manual

## Mefisto 2.7

### Volume 1: Solid Mechanics

#### Contents

1	Introduction.....	3
2	Linear Static Analysis.....	4
2.1	2-dimensional Truss.....	4
2.2	T-Junction.....	21
2.3	Membrane with a Hole.....	33
2.4	3-dimensional Truss.....	40
2.5	3-dimensional Frame.....	44
2.6	Torsion Box.....	56
2.7	Scordelis-Lo Roof.....	65
3	Normal Modes Analysis.....	78
3.1	2-dimensional Frame.....	78
3.2	2-dimensional Arc.....	85
3.3	Cantilever Beam.....	95
3.4	Fan Blade.....	101
3.5	Stiffened Plate with Point Mass.....	105
3.6	Cylindrical Box.....	115
4	Frequency Response Analysis.....	125
4.1	2-dimensional Truss.....	125
4.2	Box Beam.....	135
5	Random Response Analysis.....	163
5.1	Instrument Panel.....	163
6	Transient Response Analysis.....	183
6.1	Plane Plate.....	183

6.2 Box Beam.....	217
-------------------	-----

## 1 Introduction

The examples presented in this manual explain how to use Mefisto to analyse problems in solid mechanics. They demonstrate how to define the model, how to use the Mefisto functions and how to postprocess the results. The examples also show how to use Gmsh<sup>1</sup> for pre- and postprocessing.

The supporting files of all the examples can be found in directory `exa` of your Mefisto installation. The directory has the following subdirectories:

<code>solid</code>	<code>statresp</code>	Linear static analysis
	<code>freevib</code>	Normal modes analysis
	<code>freqresp</code>	Frequency response analysis
	<code>randresp</code>	Random response analysis
	<code>transresp</code>	Transient response analysis

1 <http://www.geuz.org/gmsh>

## 2 Linear Static Analysis

### 2.1 2-dimensional Truss

#### Summary

Directory:	exa/solid/statresp/truss2d
Objectives:	<ul style="list-style-type: none"> <li>• learn how to define a simple 2-dimensional model</li> <li>• learn how to run a linear static analysis</li> <li>• learn how to postprocess the results using the Mefisto functions</li> <li>• learn some basic postprocessing using Gmsh</li> </ul>
Dimension:	2
Elements:	<b>r2</b>
Loads:	concentrated nodal point forces
Functions:	<b>mfs_new, mfs_plot, mfs_print, mfs_stiff, mfs_statresp, mfs_getresp, mfs_results, mfs_export</b>

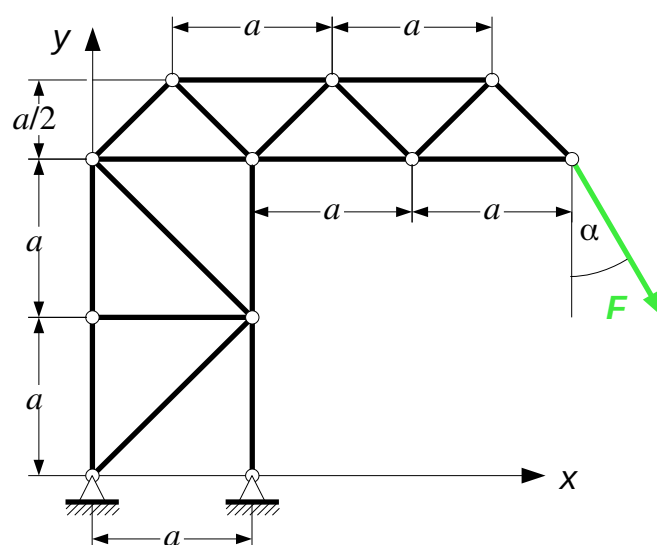


Figure 2.1-1: Truss



## Problem Description

Compute the deformation and the forces in all rods of the truss structure shown in Figure 2.1-1.

Data:  $a = 1000$  mm,  $F = 1$  kN,  $\alpha = 30^\circ$ , cross section area  $A = 400$  mm<sup>2</sup>, Young's modulus  $E = 210$  GPa

## Model Definition

First, we define the model data and open the output file. Note that consistent units have to be used for the data. In static analysis, N and mm are consistent units, and so is MPa = N/mm<sup>2</sup>.

The output file will contain some information on the model and the requested results. If errors are detected in the model definition, error messages are written to the output file. Thus, it is good practice to always have a look to the output file before postprocessing the results.

The data and the output file are defined by the following commands in file `truss_a.m`:

```
# Data

F      = 1000; % Force in N
alpha  = 30;   % Direction of force in degrees
E      = 210000; % Young's modulus in MPa
A      = 400;  % Cross section area in mm^2
a      = 1000; % Length in mm

# Output file

fid = fopen("truss_a.res", "wt");
```

Next, we define the model type and the model subtype. In solid mechanics, the model type is "**solid**", and the subtype indicates the dimension of the problem. Here, we are going to analyse a 2-dimensional truss. Thus, the subtype is "**2d**". There are no defaults for these settings.

The model type and the model subtype are defined by the fields **type** and **subtype** of the structure that defines the model. You can name this structure as you like. In this example, we name it **model**.

```
# Model type and subtype

model = struct("type", "solid", "subtype", "2d");
```

Now, we define the nodal points. The nodal points are defined by the structure **nodes** which has the fields **id** for the nodal point identifier and **coor** for

the coordinates. The nodal point identifiers and locations can be seen in Figure 2.1-2. The structure **nodes** is defined and subsequently assigned to field **nodes** of the structure **model**.

#### # Nodal points

```
nodes( 1).id = 1; nodes( 1).coor = [0, 0];
nodes( 2).id = 2; nodes( 2).coor = [a, 0];
nodes( 3).id = 3; nodes( 3).coor = [0, a];
nodes( 4).id = 4; nodes( 4).coor = [a, a];
nodes( 5).id = 5; nodes( 5).coor = [0, 2 * a];
nodes( 6).id = 6; nodes( 6).coor = [a, 2 * a];
nodes( 7).id = 7; nodes( 7).coor = [2 * a, 2 * a];
nodes( 8).id = 8; nodes( 8).coor = [3 * a, 2 * a];
nodes( 9).id = 9; nodes( 9).coor = [0.5 * a, 2.5 * a];
nodes(10).id = 10; nodes(10).coor = [1.5 * a, 2.5 * a];
nodes(11).id = 11; nodes(11).coor = [2.5 * a, 2.5 * a];

model.nodes = nodes;
```

Instead of **nodes** we could have used any other name for the structure containing the nodal point definition. However, the field of the structure **model** the definition of the nodal points is assigned to must be named **nodes**.

Before defining the elements, we define structures **geom** and **mat** containing the geometrical and material data of the elements. For rod elements, the structure **geom** has only one field that defines the area of the cross section.

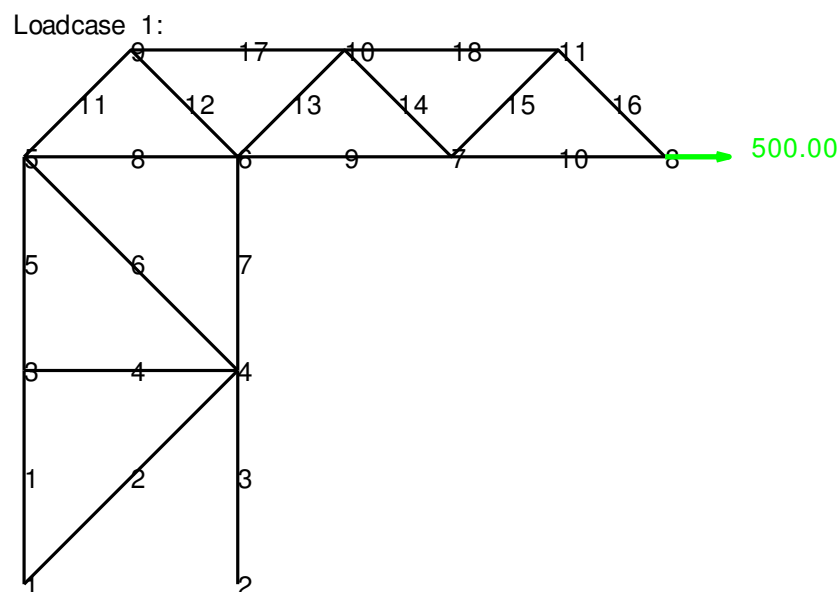


Figure 2.1-2: Truss, Finite Element Model

The structure **mat** has two fields, field **type** defining the material type and field **E** defining the modulus of elasticity. Poisson's ratio is not needed.

```
# Geometry and material

geom = struct("A", A);
mat  = struct("type", "iso", "E", E);
```

Next, we define the elements. We begin with the element identifiers and the element connectivity, i.e. the identifiers of the two nodal points the rod element is connected to. Figure 2.1-2 shows the element identifiers and the nodal point identifiers.

Subsequently, the element type, the geometrical data and the material data are assigned to each element. As all elements have the same type, geometry and material, we can do this simply by looping over all elements. Finally, we assign the structure that defines the elements to field **elements** of the structure **model**.

```
# Elements

rod( 1).id = 1; rod( 1).nodes = [ 1, 3];
rod( 2).id = 2; rod( 2).nodes = [ 1, 4];
rod( 3).id = 3; rod( 3).nodes = [ 2, 4];
rod( 4).id = 4; rod( 4).nodes = [ 3, 4];
rod( 5).id = 5; rod( 5).nodes = [ 3, 5];
rod( 6).id = 6; rod( 6).nodes = [ 4, 5];
rod( 7).id = 7; rod( 7).nodes = [ 4, 6];
rod( 8).id = 8; rod( 8).nodes = [ 5, 6];
rod( 9).id = 9; rod( 9).nodes = [ 6, 7];
rod(10).id =10; rod(10).nodes = [ 7, 8];
rod(11).id =11; rod(11).nodes = [ 5, 9];
rod(12).id =12; rod(12).nodes = [ 9, 6];
rod(13).id =13; rod(13).nodes = [ 6, 10];
rod(14).id =14; rod(14).nodes = [ 10, 7];
rod(15).id =15; rod(15).nodes = [ 7, 11];
rod(16).id =16; rod(16).nodes = [ 11, 8];
rod(17).id =17; rod(17).nodes = [ 9, 10];
rod(18).id =18; rod(18).nodes = [ 10, 11];

for n = 1 : length(rod)
    rod(n).type = "r2";
    rod(n).geom = geom; rod(n).mat = mat;
endfor

model.elements = rod;
```

Here, we have used the name **rod** for the structure that defines the elements. Again, we could have used any other name, but the definition of the elements must be assigned to field **elements** of the structure **model**.

It still remains to define the constraints and the load. There are hinges at nodal points 1 and 2, i.e. the  $x$ - and  $y$ -displacements of these nodal points must be zero. To define these hinges, we have to specify the identifiers of the nodal points and the identifiers of the degrees of freedom. The nodal point identifiers are defined in field **id** and the degree of freedom identifiers in field **dofs** of a structure we name **hinge**. This structure is subsequently assigned to the field **constraints.prescribed** of structure **model**.

To define a concentrated force, we have to specify the identifier of the nodal point the force is applied to, and the components of the force vector. The force is applied to the nodal point with identifier 8, see Figure 2.1-2. The identifier is defined in field **id** of a structure we choose to name **force**. The force vector is assigned to field **data** of the same structure. This structure is subsequently assigned to field **loads.point** of structure **model**.

```
# Constraints

hinge(1) = struct("id", 1, "dofs", [1, 2]);
hinge(2) = struct("id", 2, "dofs", [1, 2]);

model.constraints.prescribed = hinge;

# Load

force = struct("id", 8,
               "data", F * [sind(alpha), -cosd(alpha)]);

model.loads.point = force;
```

## Analysis

First, we have to create a component from the model definition stored in structure **model**. This is done by function **mfs\_new**. We name this component **truss**. In order to check if the model definition is correct we use function **mfs\_plot** to plot the model. As this is a very small model, we can switch on the plotting of nodal point and element identifiers by setting parameters **"nodid"** and **"eltid"** to 1. We also want to see the components of the load vector with their values written next to them. Thus, we set the value of parameter **"force"** to 2. Parameters **"fontsize"** and **"paperposition"** control the font size of the text in the plot on the screen and the size of the plot created with the following **print** command (in cm, if the **.octaverc** file contains the command

```
set(0, "defaultfigurepaperunits", "centimeters"); )
```

```
# Create and plot component
```

```
truss = mfs_new(fid, model);
mfs_plot(truss, "nodid", 1, "eltid", 1, "force", 2,
         "fontsize", 18, "paperposition", [0, 0, 14, 10]);
print("model.svg", "-dsvg", "-F:10");
mfs_print(fid, truss, "loads", "point");
```

Figure 2.1-2 shows the picture created by function `mfs_plot` and the `print` command.

Function `mfs_print` writes the applied load to the output file. Thus, up to now, the output file contains the following information:

Mefisto 2.7: Building new component from input "model"

```
Model Type = solid, Model Subtype = 2d

Number of nodes      =    11,  Number of elements =    18
Number of element types =    1
Number of global      degrees of freedom =    22
Number of local      degrees of freedom =    18
Number of prescribed degrees of freedom =     4
Number of dependent  degrees of freedom =     0

Number of load cases =    1
```

-----

Component "truss"

Point loads of loadcase 1

node	Fx	Fy	
8	5.000e+02	-8.660e+02	
Res.	5.000e+02	-8.660e+02	-3.598e+06

The resultant of the loads is computed with respect to the origin of the co-ordinate system.

Next, we compute the stiffness matrix, the displacements and the reaction forces. The stiffness matrix is computed by function `mfs_stiff` whereas function `mfs_statresp` computes the displacements and the reaction loads. Function `mfs_getresp` returns the strain energy and the energy of the residual forces. The latter should be small compared to the strain energy.

Function `mfs_print` writes the displacements and the reaction forces to the output file. Function `mfs_plot` generates a freebody diagram of the deformed truss, showing both the applied forces and the reaction forces. Figure 2.1-3 shows the picture created by the `print` command.

```
# Compute displacements
```

```
truss = mfs_stiff(truss);
truss = mfs_statresp(truss);
ES = mfs_getresp(truss, "statresp", "strnegy");
WR = mfs_getresp(truss, "statresp", "workrsl");
```

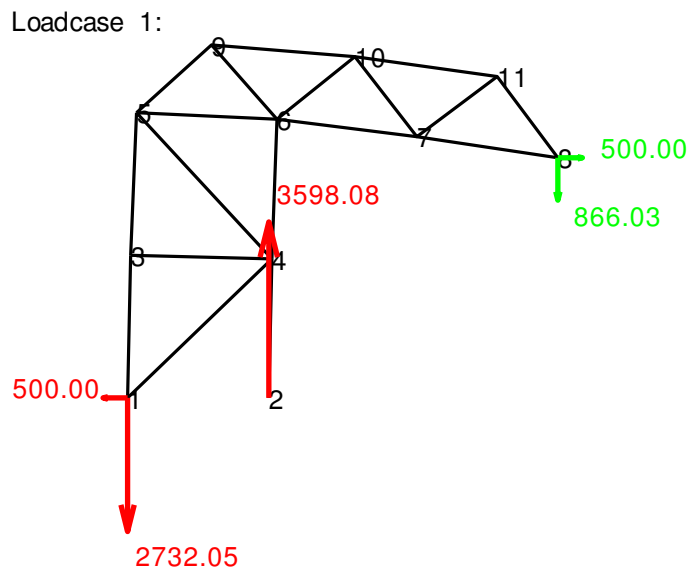


Figure 2.1-3: Truss, Freebody Diagram of Deformed Truss

```
fprintf(fid, "Strain energy    : %10.3e\n", ES);
fprintf(fid, "Work of residual: %10.3e\n", WR);

mfs_print(fid, truss, "statresp", "disp", "reac");
mfs_plot(truss, "deform", 1, "nodid", 1, "force", 2,
         "reac", 2, "fontsize", 18);
print("freebody.svg", "-dsvg", "-F:10");
```

This section of the code adds the following information to the output file:

```
Strain energy    : 3.860e+02
Work of residual: -1.766e-12
```

-----

Component "truss"

Displacements of loadcase 1

node	ux	uy
1	0.000e+00	0.000e+00
2	0.000e+00	0.000e+00
3	5.967e-02	2.657e-02
4	5.967e-02	-4.283e-02
5	1.725e-01	5.314e-02
6	1.578e-01	-7.376e-02
7	1.328e-01	-4.103e-01
8	1.285e-01	-8.174e-01
9	2.578e-01	-2.976e-03
10	2.990e-01	-2.295e-01
11	3.196e-01	-6.116e-01

Reaction loads of loadcase 1

node	Fx	Fy
1	-5.000e+02	-2.732e+03
2	0.000e+00	3.598e+03

```
Res.      -5.000e+02   8.660e+02   3.598e+06
```

The resultant of the reaction loads refers to the origin of the coordinate system.

Next, we compute the element results, i.e. the forces, the stresses and the strains in the rods, and write them to the output file. Function `mfs_results` computes the element results, and function `mfs_print` writes them to the output file. The forces in the rods are stress resultants. Thus, they are requested with the keyword `"resultant"`.

```
# Compute element results

truss = mfs_results(truss, "statresp", "element");
mfs_print(fid, truss, "statresp",
          "resultant", "stress", "strain");
```

The following information is added to the output file:

```
-----
Component "truss"
```

```
Stress resultants of loadcase 1:
```

```
Rod elements:
```

element	N
1	2.232e+03
2	7.071e+02
3	-3.598e+03
4	5.829e-13
5	2.232e+03
6	-7.071e+02
7	-2.598e+03
8	-1.232e+03
9	-2.098e+03
10	-3.660e+02
11	2.449e+03
12	-2.449e+03
13	-1.225e+03
14	1.225e+03
15	-1.225e+03
16	1.225e+03
17	3.464e+03
18	1.732e+03

```
Stresses of loadcase 1:
```

```
Rod elements:
```

element	stress
1	5.580e+00
2	1.768e+00
3	-8.995e+00
4	1.457e-15
5	5.580e+00
6	-1.768e+00
7	-6.495e+00
8	-3.080e+00
9	-5.245e+00
10	-9.151e-01

```

11    6.124e+00
12   -6.124e+00
13   -3.062e+00
14    3.062e+00
15   -3.062e+00
16    3.062e+00
17    8.660e+00
18    4.330e+00

```

Strains of loadcase 1:

Rod elements:

element	strain
1	2.657e-05
2	8.418e-06
3	-4.283e-05
4	6.939e-21
5	2.657e-05
6	-8.418e-06
7	-3.093e-05
8	-1.467e-05
9	-2.498e-05
10	-4.357e-06
11	2.916e-05
12	-2.916e-05
13	-1.458e-05
14	1.458e-05
15	-1.458e-05
16	1.458e-05
17	4.124e-05
18	2.062e-05

Finally, we export the model as well as the forces and stresses in the rods for postprocessing with Gmsh. The keyword "**msh**" indicates that we want to export the model as a Gmsh MSH ASCII file (the only file format currently supported).

The keyword "**mesh**" requests the export of the finite element mesh. Note that only nodal points and elements are exported.

The keyword "**statresp**" requests the export of the results of a static analysis. The following keywords "**resultant**" and "**stress**" specify that we want to export the stress resultants, i.e. the forces in the rods, and the stresses.

```

# Export results to Gmsh

mfs_export("truss2d.msh", "msh", truss, "mesh");
mfs_export("truss2d.pos", "msh", truss,
          "statresp", "resultant", "stress");

fclose(fid);

```

The last command, **fclose**, closes the output file.



## Postprocessing

Some limited postprocessing can be done using the Mefisto functions. In this simple example, we used the `mfs_plot` command to generate a freebody diagram of the deformed truss. You may perhaps have noticed that the time needed for plotting is by far larger than the time needed for the computations. Thus, for larger models, it is more convenient to use Gmsh for postprocessing, as we will do now to have a look at the forces in the rods.

Once you have started Gmsh, import the finite element mesh. To do so, click on `File` in the main window and select `Open...` in the menu that opens (see Figure 2.1-4). In the file selector menu that appears, select file `truss_a.msh` and click the OK button (see Figure 2.1-5).

Actually, you do not yet see the mesh. To make it visible, you have to set some options. In the main window, click on `Tools` and select `Options` (see Figure 2.1-6). In the menu that appears click on `Mesh` and activate the visibility of the nodes and the lines. Also switch on node labels and element labels (see Figure 2.1-7). Next, go to the `Color` tab and select coloring mode “By element type” (see Figure 2.1-8). Then, go to the `Aspect` tab and increase the line width (see Figure 2.1-9). Now, the picture on the screen should look like Figure 2.1-10.

Next, you have to import the results. The file containing the results is merged with the mesh. Thus, to import the results, click on `File` and select `Merge...`



Figure 2.1-4: Gmsh: Open the Mesh File

In the file selector menu (see Figure 2.1-5), select file `truss_a.pos`. Figure 2.1-11 shows what you should see now on the screen. In the tree on the left side, an item called `Post-processing` is added, with two subitems, so

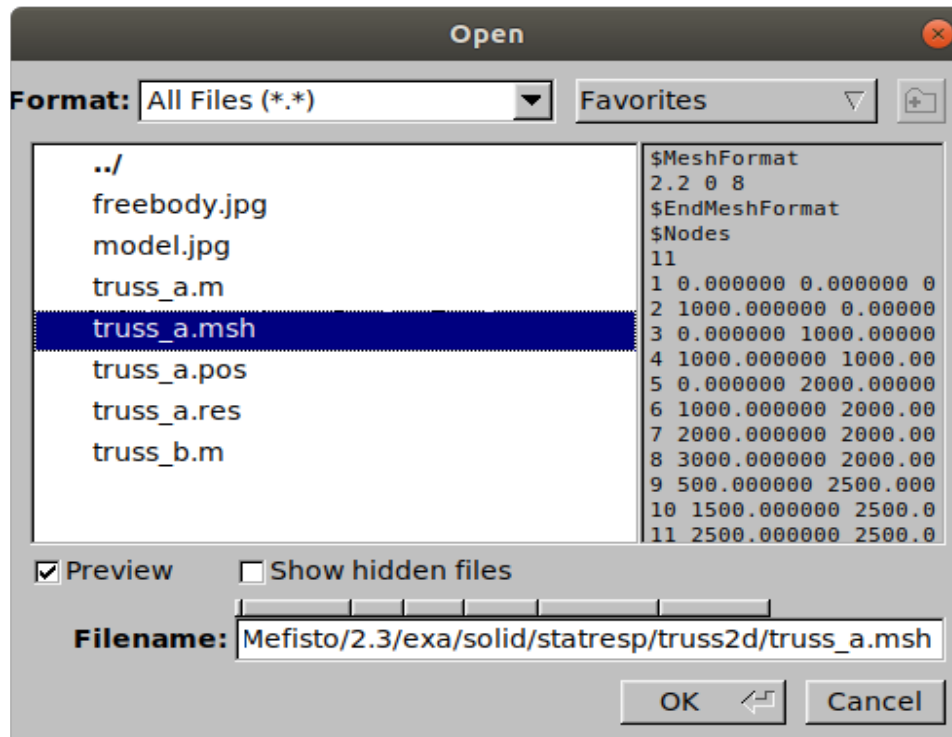


Figure 2.1-5: Gmsh: Input File Selector

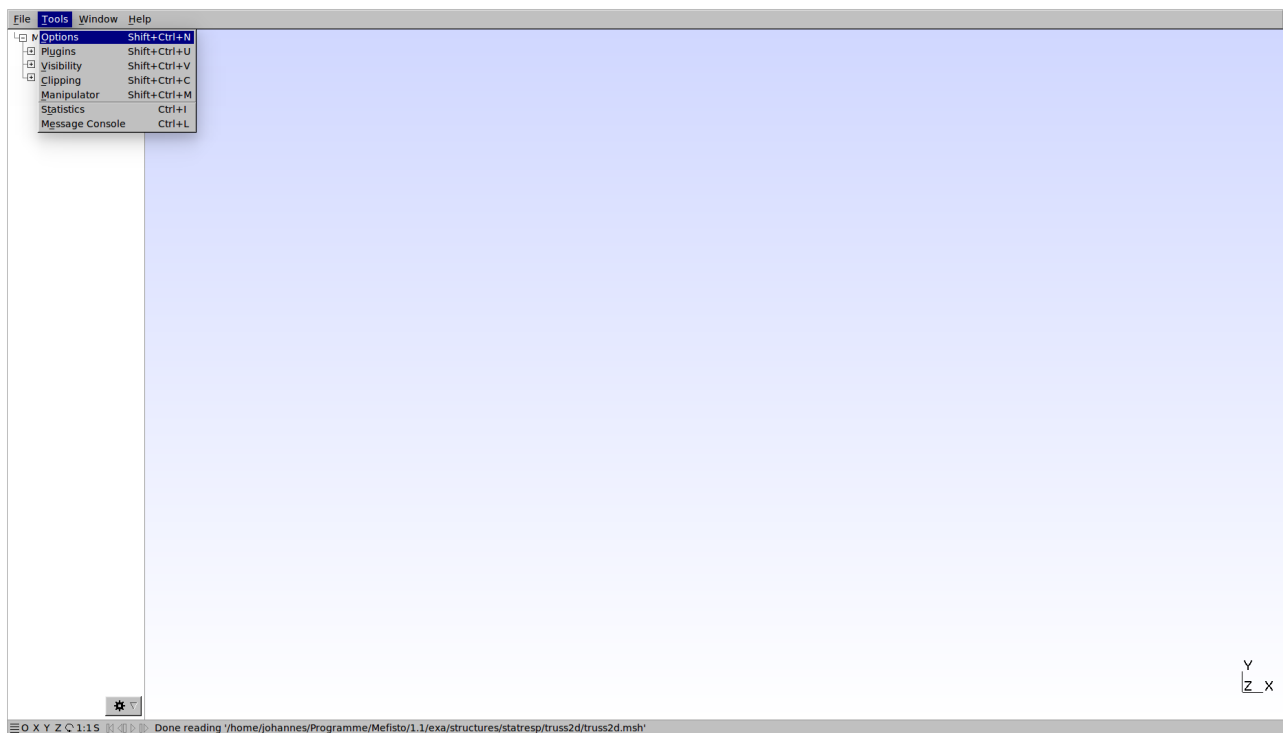


Figure 2.1-6: Gmsh: Accessing the Options

called views that are both active. The first view contains the stress resultants

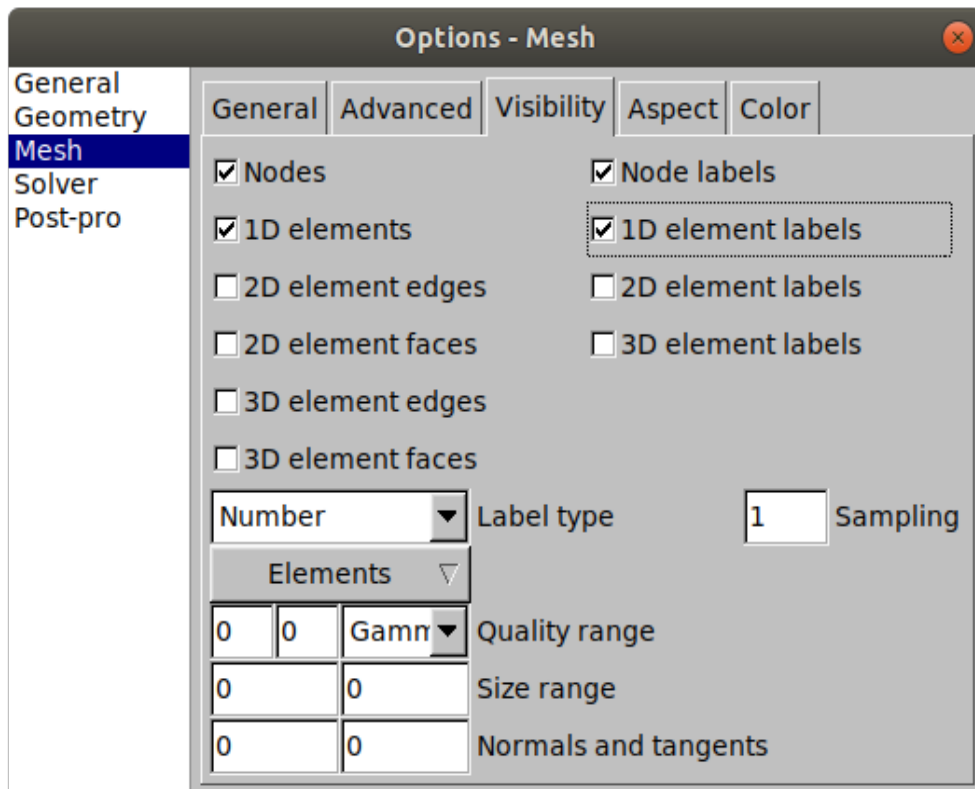


Figure 2.1-7: Gmsh: Mesh Visibility Tab

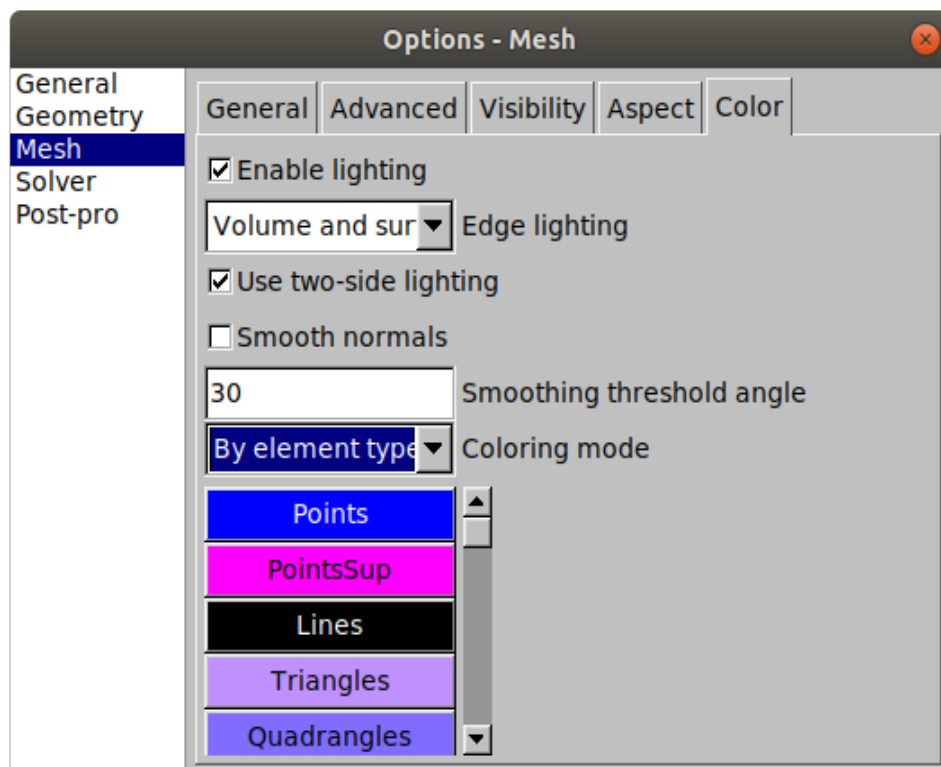


Figure 2.1-8: Gmsh: Mesh Color Tab

and the second the stresses.

To produce a colour plot of the stress resultants, first switch off the visibility of

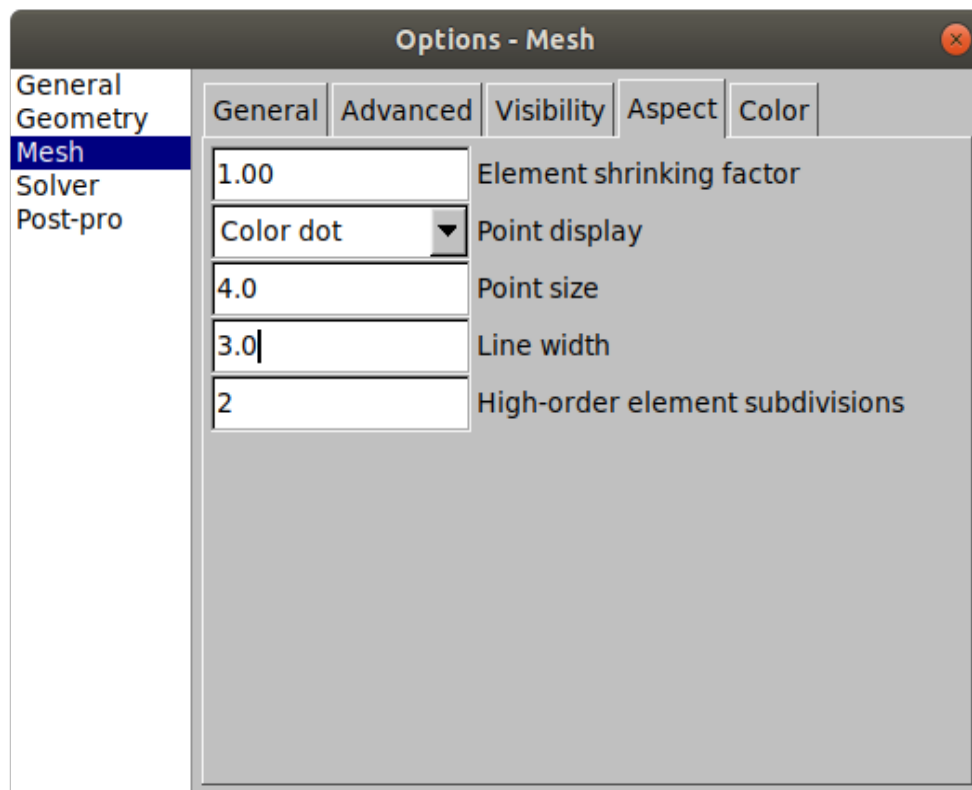


Figure 2.1-9: Gmsh: Mesh Aspect Tab

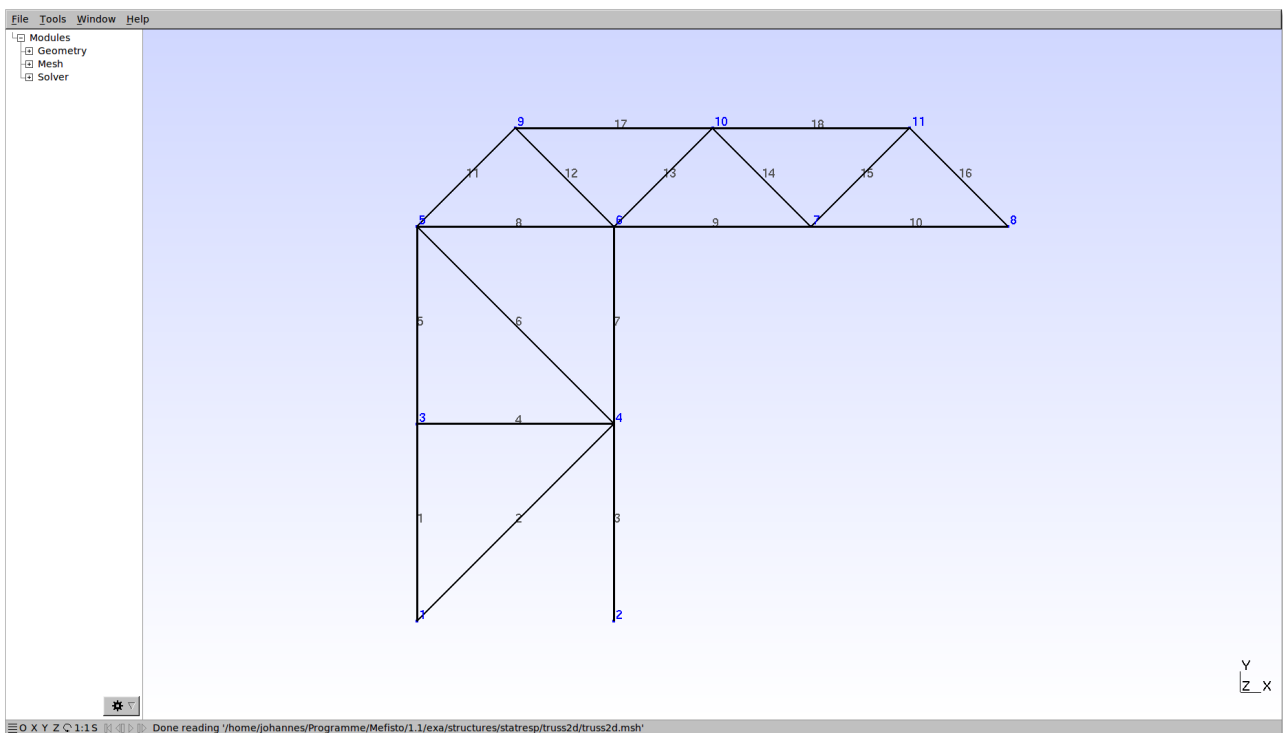


Figure 2.1-10: Gmsh: Mesh Plot

the mesh (**Tools** → **Options**) and deactivate the second view. Next, start the View Options menu. Click on the triangle on the right side of the first view entry and select **Options** (see Figure 2.1-12). This brings up the menu shown in Figure 2.1-13.

In the General tab, select Filled iso-values as intervals type and manually define the range from -4000 to 3500. Unfortunately, the default is not very useful in this case because Gmsh assumes you want to look at von Mises stresses that are always positive.

Element results in Gmsh are always interpreted as the nine components of the stress tensor. Mefisto stores the forces of the rods as the first component, the remaining eight components being all zero. To display the first component, go to the Visibility tab, select Force Scalar at the bottom of the menu and select component 0 (see Figure 2.1-14). Notice that Gmsh numbers the nine components from 0 to 8.

Finally, go to the Aspect tab (see Figure 2.1-15) and select 3D cylinder as Line display. Figure 2.1-16 shows what you should see now.

As this is a small model, it is also useful to write the values of the forces in the rod elements to the plot. To display these values, go to the General tab and select Numeric values as intervals type. Then, only the numbers are displayed. To see the elements as well, go to the Visibility tab and switch on

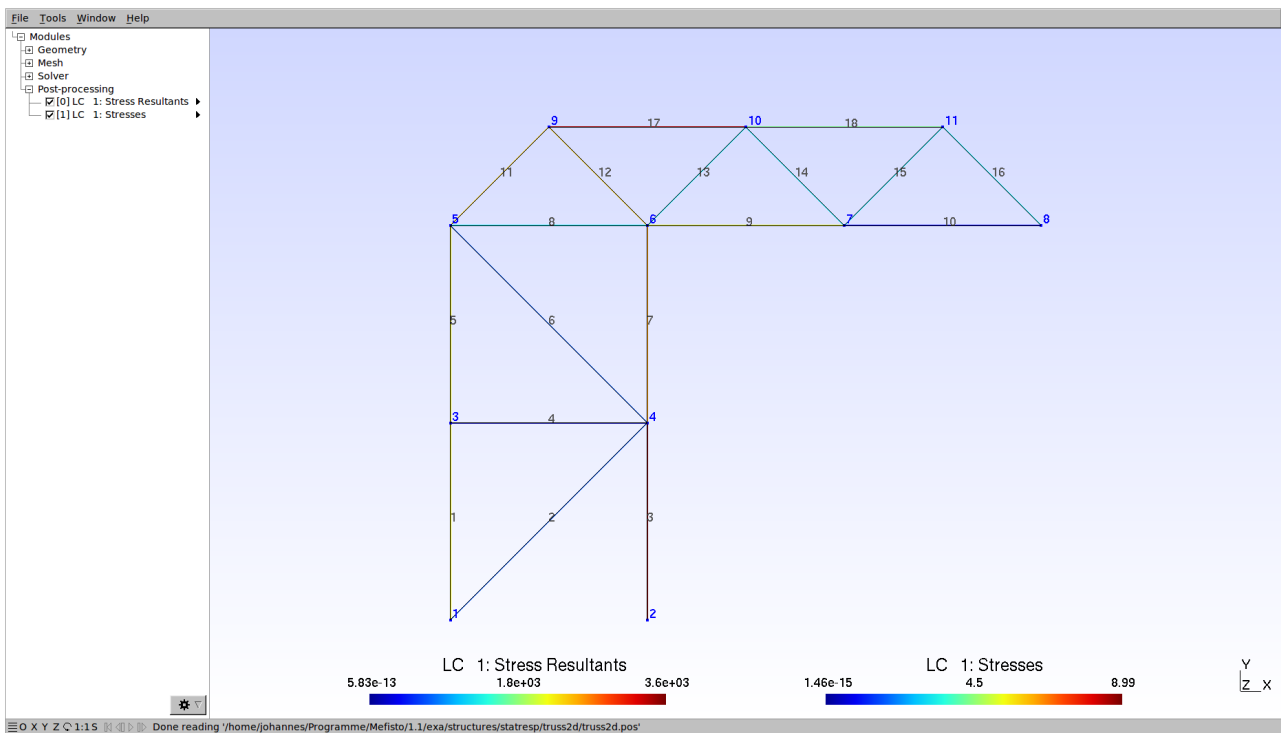


Figure 2.1-11: Gmsh: Screen after Merging the Results

Draw elements outline. If you want, you can switch off Show value scale. Then, you should get Figure 2.1-17.

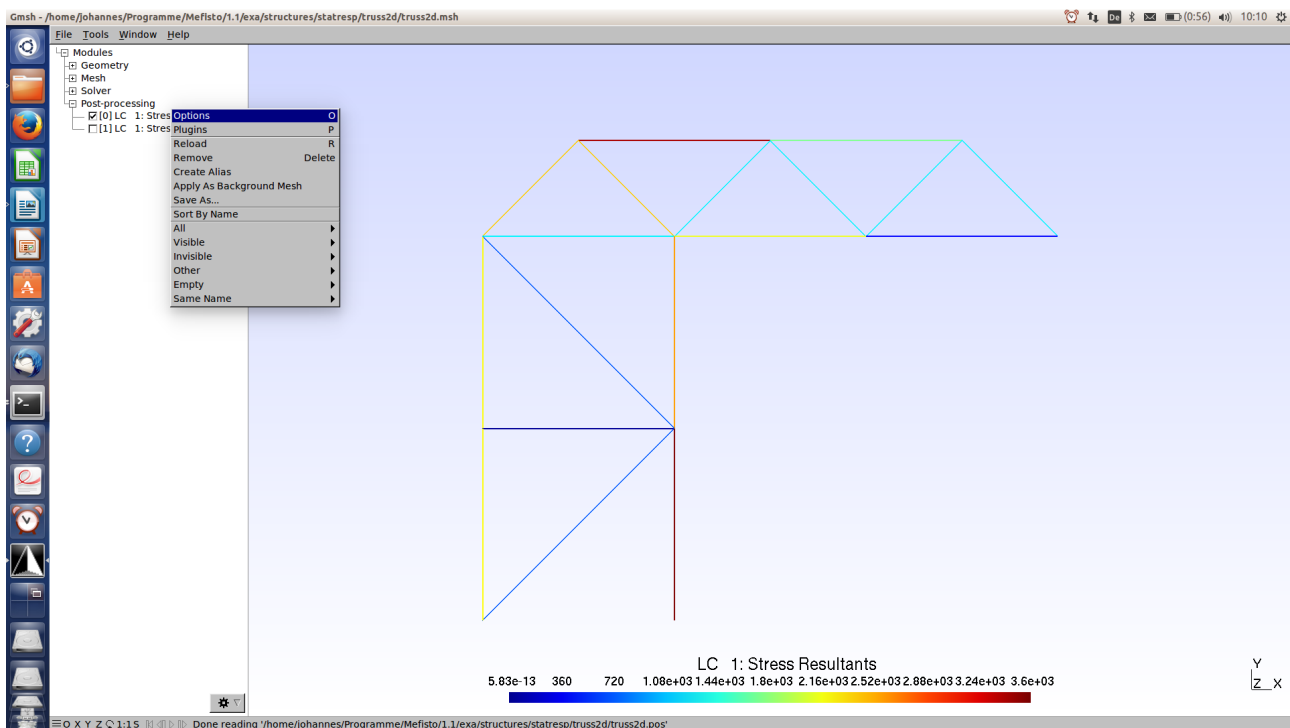


Figure 2.1-12: Gmsh: Starting the View Options Menu

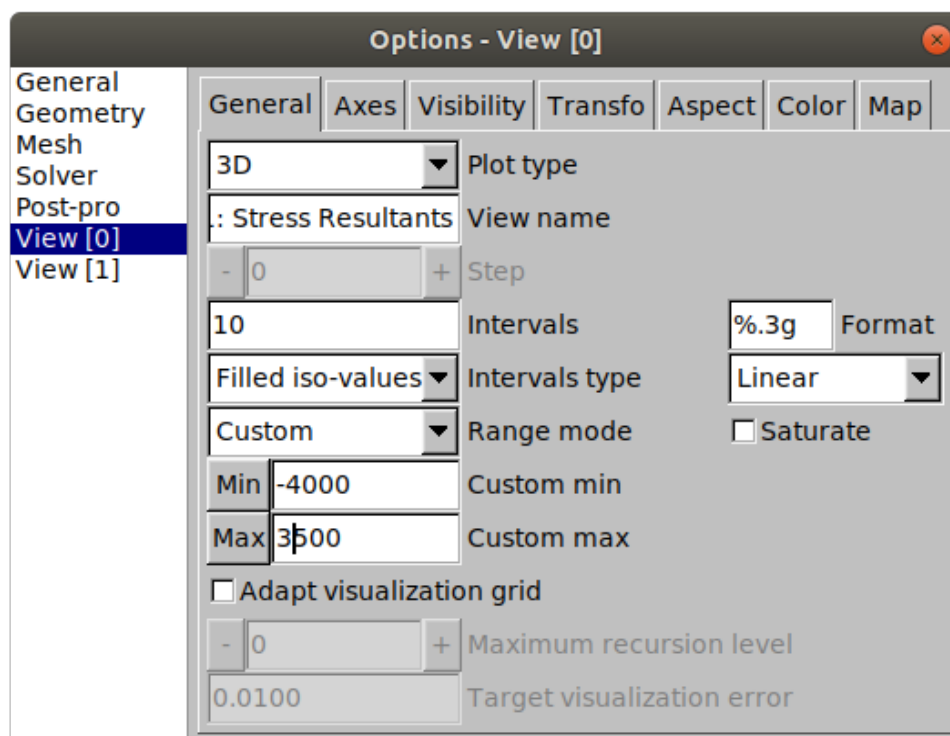


Figure 2.1-13: Gmsh: View General Tab

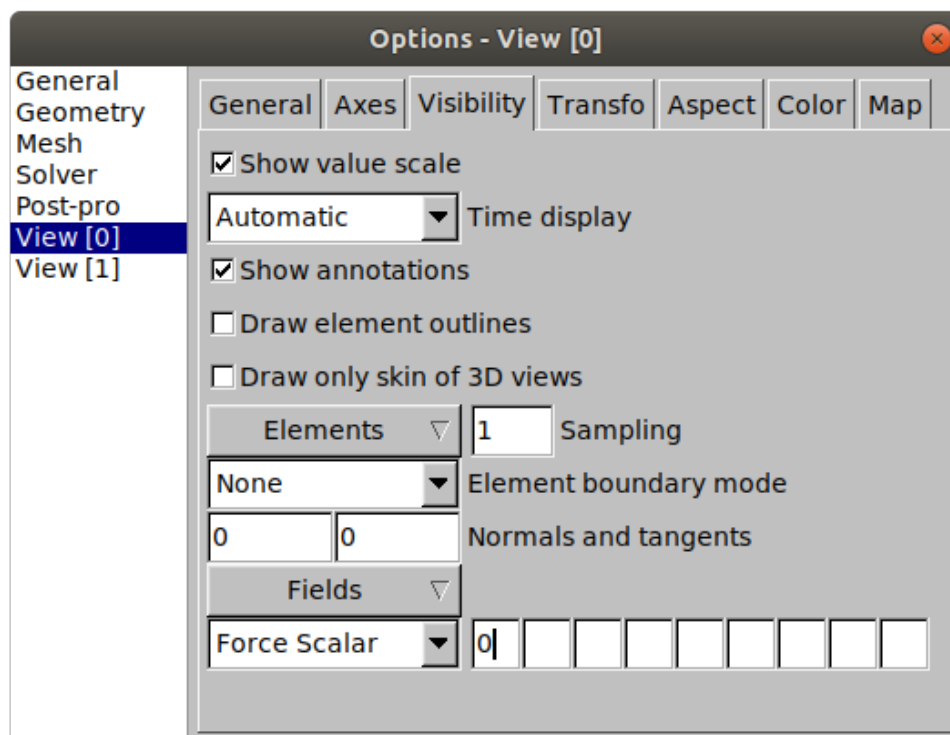


Figure 2.1-14: Gmsh: View Visibility Tab

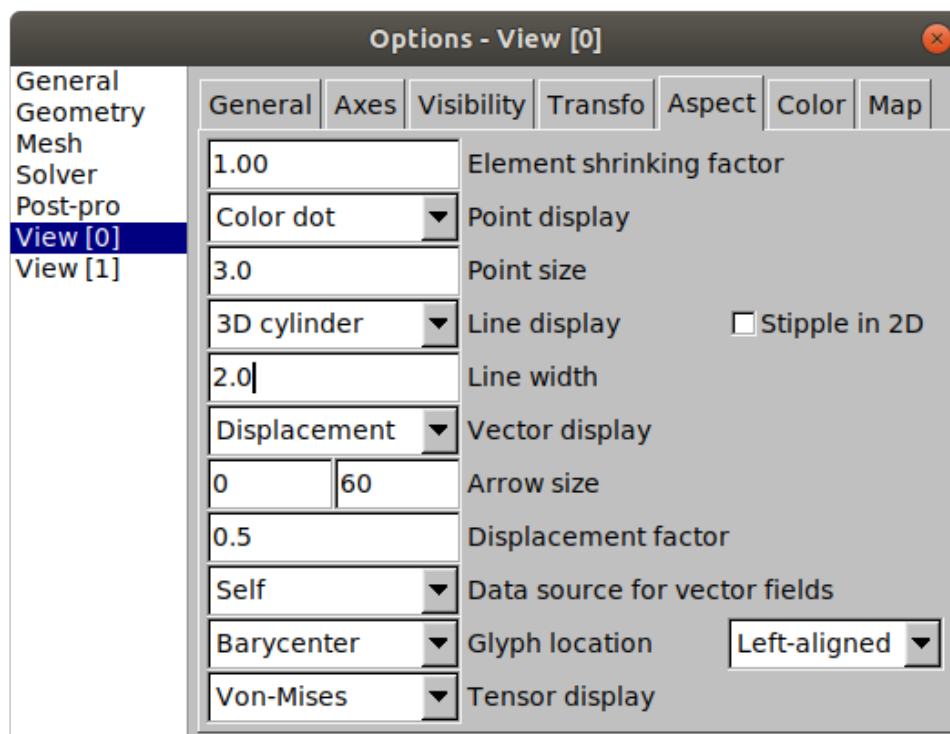


Figure 2.1-15: Gmsh: View Aspect Tab

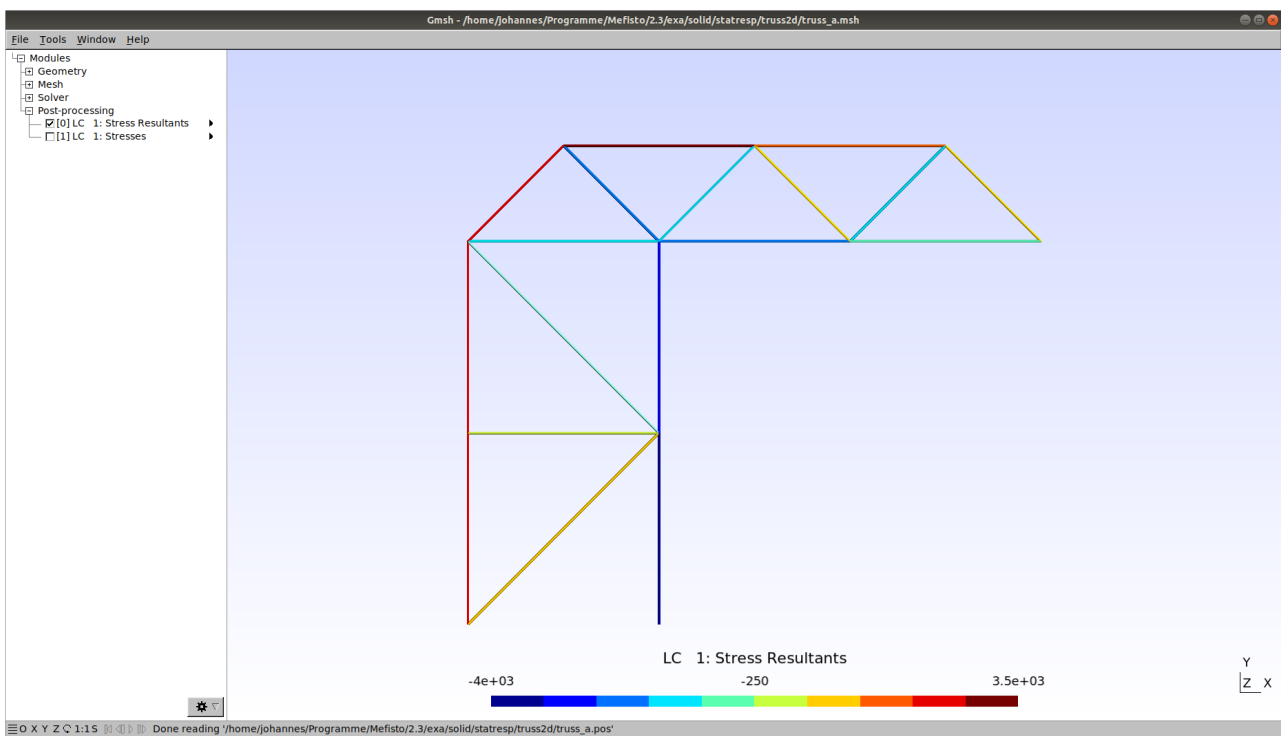


Figure 2.1-16: Gmsh: Colour Plot of Rod Forces

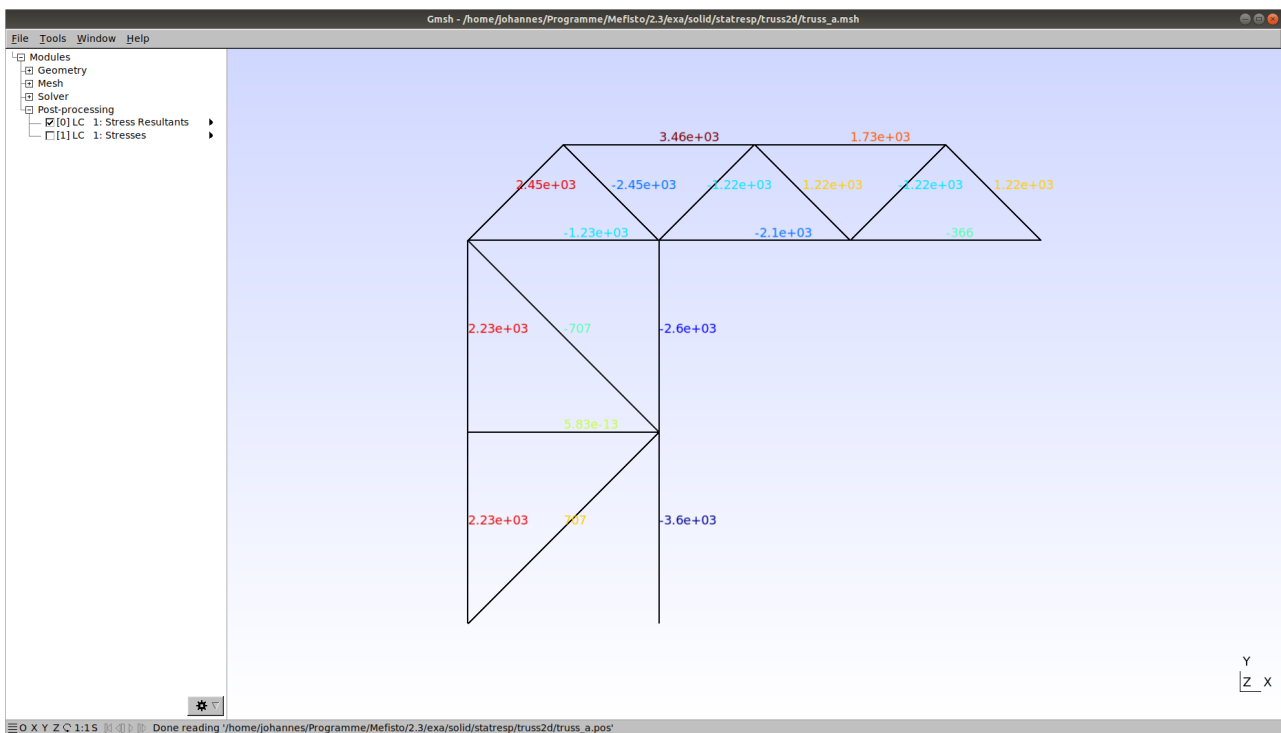


Figure 2.1-17: Gmsh: Forces in the Rods



## 2.2 T-Junction

### Summary

Directory:	exa/solid/statresp/junction2d
Objectives:	<ul style="list-style-type: none"> <li>• learn how to define the geometry of a 2-dimensional structure using Gmsh</li> <li>• learn how to define an unstructured 2-dimensional mesh using Gmsh</li> <li>• learn how to define the data needed to build a Mefisto model from a Gmsh MSH file</li> <li>• learn how to run a linear static analysis</li> <li>• learn how to postprocess results using Gmsh</li> <li>• study the effect of using different element types and different mesh sizes</li> </ul>
Dimension:	2
Elements:	t3, q4, t6, q8, q9, t3r, q4r
Loads:	prescribed displacements
Functions:	mfs_import, mfs_new, mfs_print, mfs_stiff, mfs_statresp, mfs_results, mfs_export

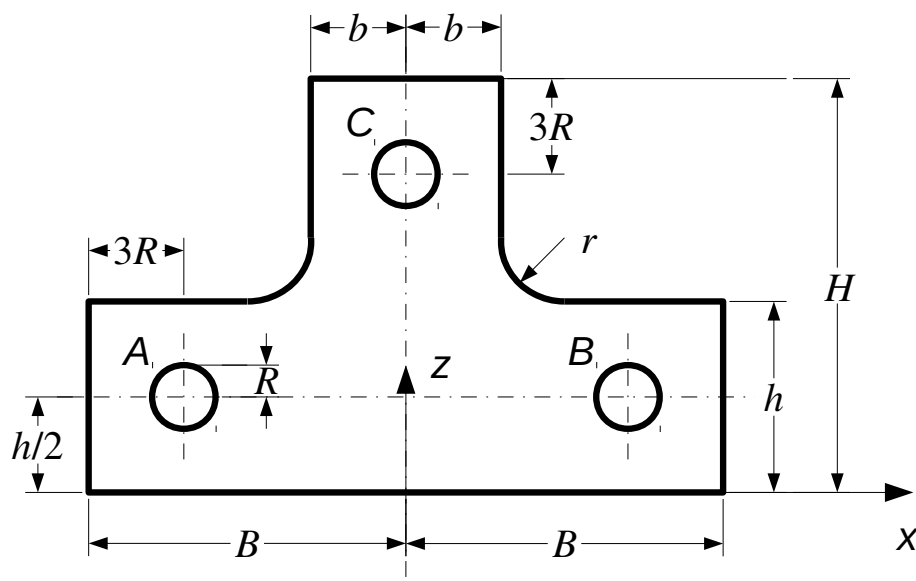


Figure 2.2-1: T-Junction

## Problem Description

Compute the deformation and the stresses of the T-junction shown in Figure 2.2-1. The structure is clamped at the edges of the holes *A* and *B*. At the edge of hole *C*, a displacement  $u_P$  in x-direction is prescribed.

Data:  $B = 50$  mm,  $b = 10$  mm,  $H = 60$  mm,  $h = 20$  mm,  $r = 5$  mm,  $R = 2$  mm, thickness  $t = 2$  mm, Young's modulus  $E = 210$  GPa, Poisson's ratio  $\nu = 0.3$ ,  $u_P = 0.1$  mm

## Model Definition

In this example, we use Gmsh to define the geometry and the mesh. We could do this interactively using the Gmsh GUI, but we prefer to use a command file. With a command file, we can parametrize the model. So, once we have set up the command file, it is easy to play with the various parameters.

The commands to define the geometry and the mesh are contained in file `junction.geo`. The file starts with a definition of the dimensions:

```
// Dimensions [mm]

DefineConstant [ B = 50,
                 b = 10,
                 H = 60,
                 h = 20,
                 r = 5,
                 R = 2 ];
```

Next, the meshing parameters are defined. Gmsh will ask for the number of elements along  $2B$  and the type of element to be used. You can select between linear and quadratic elements, between triangles and quadrangles and between complete and incomplete quadratic elements.

```
// Meshing parameters

nelb = GetValue("Number of Elements along Width?", 50);
es = 2 * B / nelb;
ef = 0.5 * es;
Mesh.ElementOrder =
GetValue("Element Order: 1 = Linear, 2 = Quadratic?", 1);
If (Mesh.ElementOrder == 2)
    Mesh.SecondOrderIncomplete =
    GetValue("Incomplete Order: 0 = no, 1 = yes?", 0);
EndIf
Mesh.RecombineAll =
GetValue("Create Quadrangles: 0 = no, 1 = yes?", 1);

Mesh.Smoothing = 5;
```

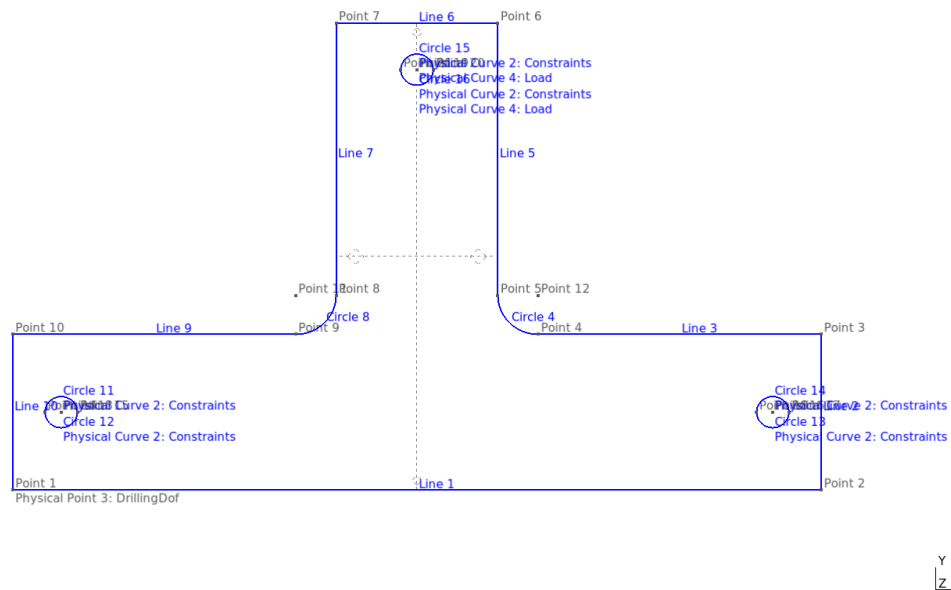


Figure 2.2-2: T-Junction, Geometry

Now, the outer contour is defined. First, we define the points on the contour and the centres of the two radii. Subsequently, we can define the straight lines and the circles which are eventually combined to build a so called line loop. The identifiers of the points and the lines can be seen in Figure 2.2-2.

```
// Geometry: Outer Contour
```

```
Point( 1) = {-B, 0, 0, es};
Point( 2) = { B, 0, 0, es};
Point( 3) = { B, h, 0, es};
Point( 4) = { b + r, h, 0, es};
Point( 5) = { b, h + r, 0, es};
Point( 6) = { b, H, 0, es};
Point( 7) = {-b, H, 0, es};
Point( 8) = {-b, h + r, 0, es};
Point( 9) = {-b - r, h, 0, es};
Point(10) = {-B, h, 0, es};

Point(11) = {-b - r, h + r, 0}; // Center of left circle
Point(12) = { b + r, h + r, 0}; // Center of right circle

Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};

Circle(4) = {4, 12, 5};

Line(5) = {5, 6};
Line(6) = {6, 7};
Line(7) = {7, 8};
```

```

Circle(8) = {8, 11, 9};

Line(9) = {9, 10};
Line(10) = {10, 1};

Line Loop(1) = {1 : 10};

```

The geometry of the holes is defined in the same way:

```

// Geometry: Holes

Point(13) = {-B + 3 * R, h/2, 0};
Point(14) = {-B + 2 * R, h/2, 0, ef};
Point(15) = {-B + 4 * R, h/2, 0, ef};

Circle(11) = {15, 13, 14};
Circle(12) = {14, 13, 15};

Line Loop(2) = {11, 12};

Point(16) = {B - 3 * R, h/2, 0};
Point(17) = {B - 2 * R, h/2, 0, ef};
Point(18) = {B - 4 * R, h/2, 0, ef};

Circle(13) = {18, 16, 17};
Circle(14) = {17, 16, 18};

Line Loop(3) = {13, 14};

Point(19) = {0, H - 3 * R, 0};
Point(20) = {R, H - 3 * R, 0, ef};
Point(21) = {-R, H - 3 * R, 0, ef};

Circle(15) = {20, 19, 21};
Circle(16) = {21, 19, 20};

Line Loop(4) = {15, 16};

```

Now, we can define the surface. The first entry is the identifier of the line loop of the outer contour. The following identifiers define the line loops of the holes. The **Physical Surface** command is used to give the surface the name "**Junction**" which will be used later to assign the thickness and the material data to this surface.

```

// Geometry: Surface

Plane Surface(1) = {1, 2, 3, 4};
Physical Surface ("Junction") = {1};

```

Likewise, the **Physical Line** commands are used to assign names to the

holes that will be used later to assign the constraints and the prescribed displacements. The same constraints are assigned to all holes but only the upper hole is given prescribed displacements.

```
// Constraints

Physical Line("Constraints") = {11 : 16};
Physical Point("DrillingDof") = {1};

// Load

Physical Line("Load") = {15, 16};
```

The remaining commands in the file, not shown here, add some annotations to the plot. These annotations are displayed if the postprocessing view **"Point Labels"** is activated.

To generate the mesh, start Gmsh and import file **junction.geo** by clicking **File → Open...**. Answer the questions in the menus that open. Then, expand the **Mesh** menu in the tree on the left by clicking on the + symbol left to it, and click on **2D** (see Figure 2.2-3). You can control the visibility of the mesh using the **Options** menu (**Tools → Options**). To store the mesh, click on **Save** at the end of the **Mesh** menu. This will create a file **junction.msh** containing the mesh data.

Under Linux there is a faster way to generate the mesh. Just type

```
gmsh junction.geo -2
```

and answer the questions.

A typical mesh can be seen in Figure 2.2-4.

File **junction.msh** contains the definition of the nodal points and elements. All other data, like the thickness, the material data, the constraints and the loads have to be defined by the translation data. The definition of these data is based on the physical groups.

The translation data are defined in the GNU Octave script used for the analysis, i.e. in file **junction.m**. The file starts with the definition of the data:

```
# Data (N, mm)

E = 210000; % Young's modulus
ny = 0.3; % Poisson's ratio
t = 2; % Thickness
u = 0.1; % Prescribed displacement of upper hole
```

The data are followed by a list that relates the Gmsh element types to the Mefisto element types:

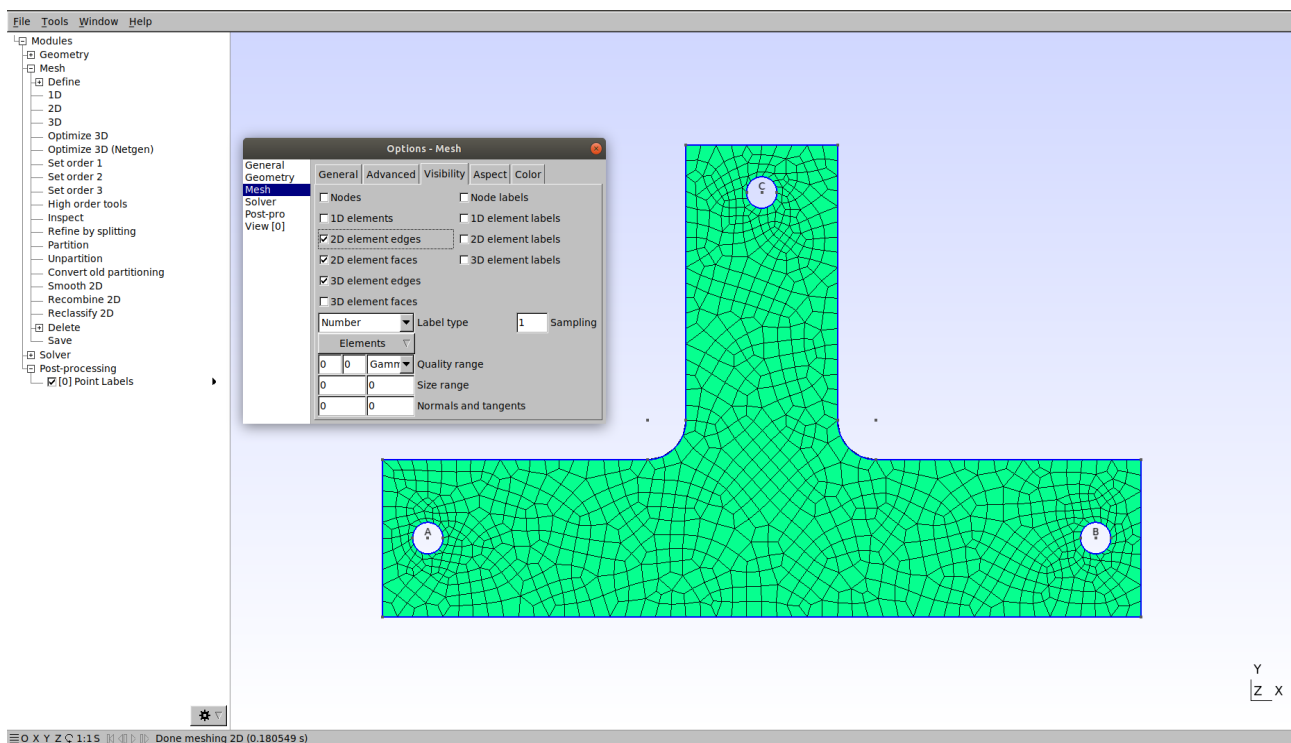


Figure 2.2-3: Gmsh: Generating the Mesh

```
name = {"2 = t3", "3 = q4", "9 = t6", "10 = q9", "16 = q8"};
```

Each string connects a Gmsh element type (left) with the name of a Mefisto element (right). If all elements in a physical group have the same type, it is sufficient to specify the Mefisto element type only.

Next, the output file is opened:

```
# Output file

fid = fopen("junction.res", "wt");
```

The translation data are defined in a structure we call **td**. We begin with defining the type and the subtype of the model:

```
# Translation data

td = struct("type", "solid", "subtype", "2d");
```

Next, we define the geometrical and the material data:

```
geom = struct("t", t);
mat = struct("type", "iso", "E", E, "ny", ny);
```

These data are now assigned to the elements of the physical group **Junction**. We do so by adding a field with name **Junction** to the structure **td**:

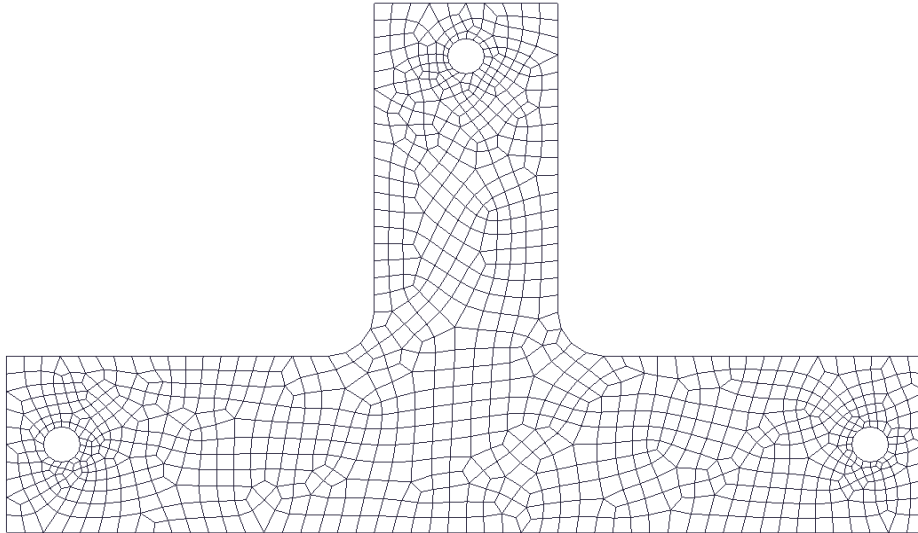


Figure 2.2-4: T-Junction, Typical Mesh

```
td.Junction = struct("type", "elements",
                    "name", {name},
                    "geom", geom,
                    "mat", mat);
```

Because variable **name** is a cell array, it has to be enclosed by curly brackets. Otherwise, GNU Octave will create a structure array.

The constraints are defined in the same way. The nodal points at which the constraints have to be applied are contained in the physical group **Constraints**. Thus, the constraint data are supplied in a field with name **Constraints**:

```
td.Constraints = struct("type", "constraints",
                       "name", "prescribed",
                       "dofs", [1, 2]);
```

Finally, field **Load** assigns the prescribed displacements to the nodal points in the physical group **Load**:

```
td.Load = struct("type", "loads",
                 "name", "disp",
                 "data", [u, 0]);
```

This completes the definition of the model.

## Analysis

First, we use function `mfs_import` to generate a Mefisto model description from the mesh file and the translation data. Then, we continue the analysis using the same functions as in Example 2.1.

```
# Import msh file

model = mfs_import(fid, "junction.msh", "msh", td);

# Create component

junction = mfs_new(fid, model);

# Analysis

junction = mfs_stiff(junction);
junction = mfs_statresp(junction);
junction = mfs_results(junction, "statresp", "element");

mfs_print(fid, junction, "statresp", "reac");

# Export results to Gmsh

mfs_export("junction.pos", "msh", junction,
           "statresp", "disp", "reac", "stress");
```

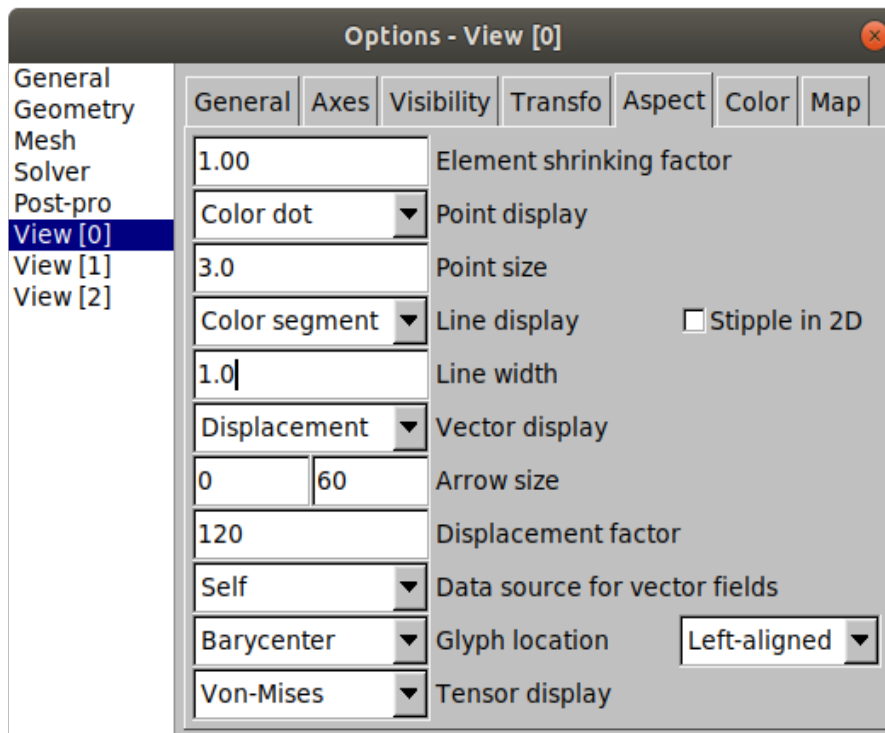


Figure 2.2-5: Gmsh: View Aspect Tab



```
fclose(fid);
```

File `junction.pos` contains displacements, reaction loads and stresses.

## Postprocessing

After opening file `junction.msh` and merging file `junction.pos`, three views can be seen in Gmsh, entitled “Deformation”, “Reac. Loads” and “Stresses”.

To generate a picture showing the deformed structure, uncheck all views but the one entitled “Deformation” and switch off the visibility of the undeformed mesh (Tools → Options → Mesh → Visibility). Then, open the options of view 0. First, go to the Aspect tab and make sure Vector Display is Displacement (see Figure 2.2-5). You can play with the Displacement factor to increase or decrease the deformation. To see the mesh, go to the Visibility tab and check Draw Element Outlines (see Figure 2.2-6). This results in the picture shown in Figure 2.2-8. This picture was created by File → Export ... . The file format is defined by the extension `.jpg` (see Figure 2.2-7).

To generate a picture showing the von Mises stress, uncheck all views but the one entitled “Stresses”. The settings needed to obtain the picture shown in Figure 2.2-9 can be seen in Figures 2.2-5 and 2.2-6.

Finally, to generate a picture showing the principal stress trajectories, go to

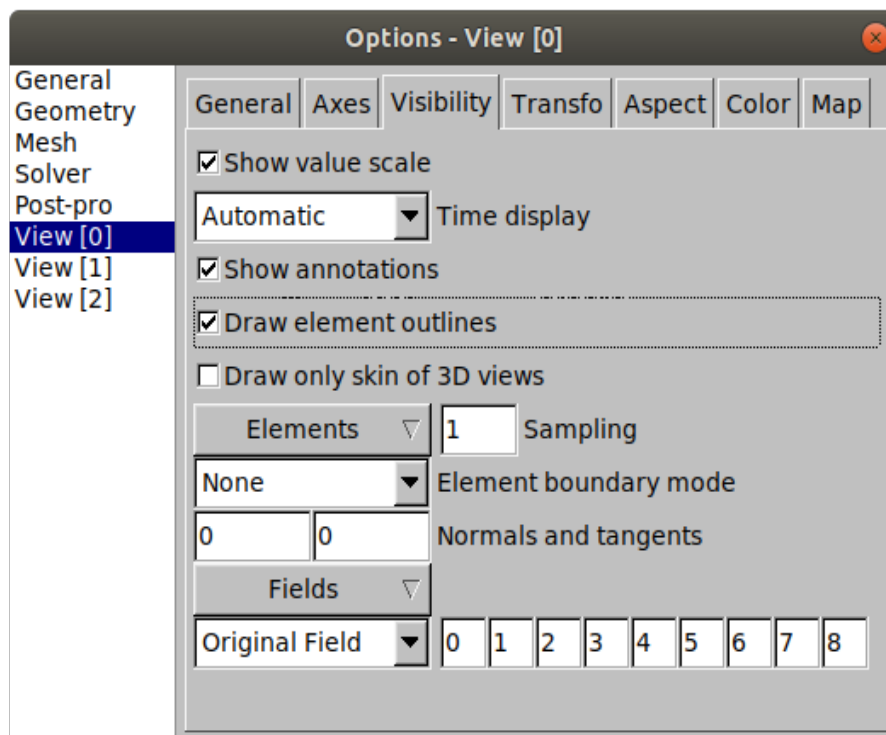


Figure 2.2-6: Gmsh: View Visibility Tab

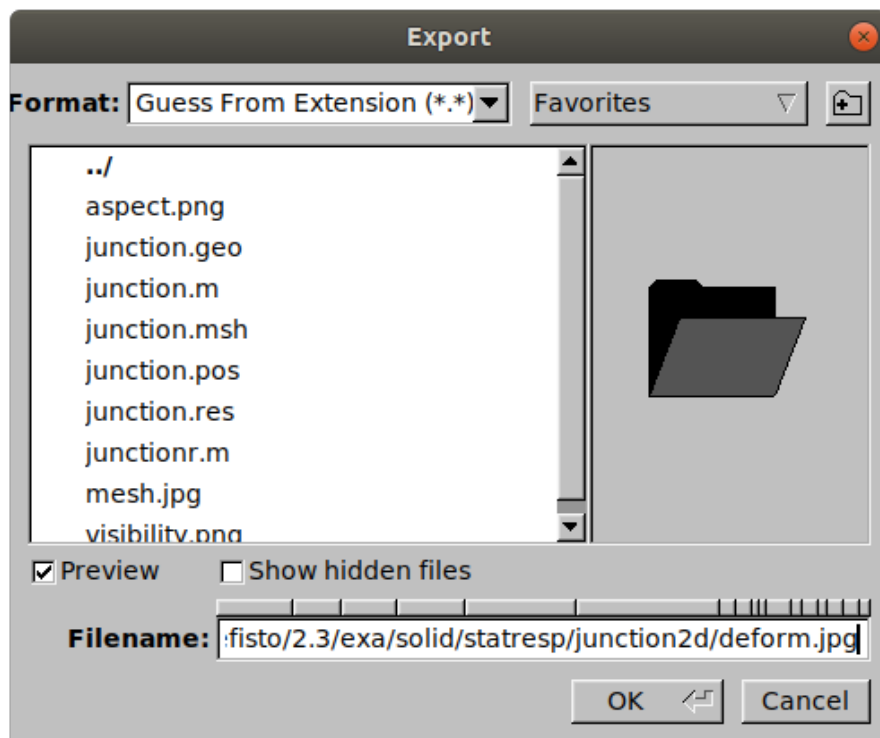


Figure 2.2-7: Gmsh: File → Export Menu

the Aspect tab and select Tensor Display Eigen Vectors and Vector Display Line (see Figure 2.2-10). This will generate the picture shown in Figure 2.2-11. The line segments are tangent to the principal stress trajectories.

The results have been obtained with the proposed default values of the mesh parameters.

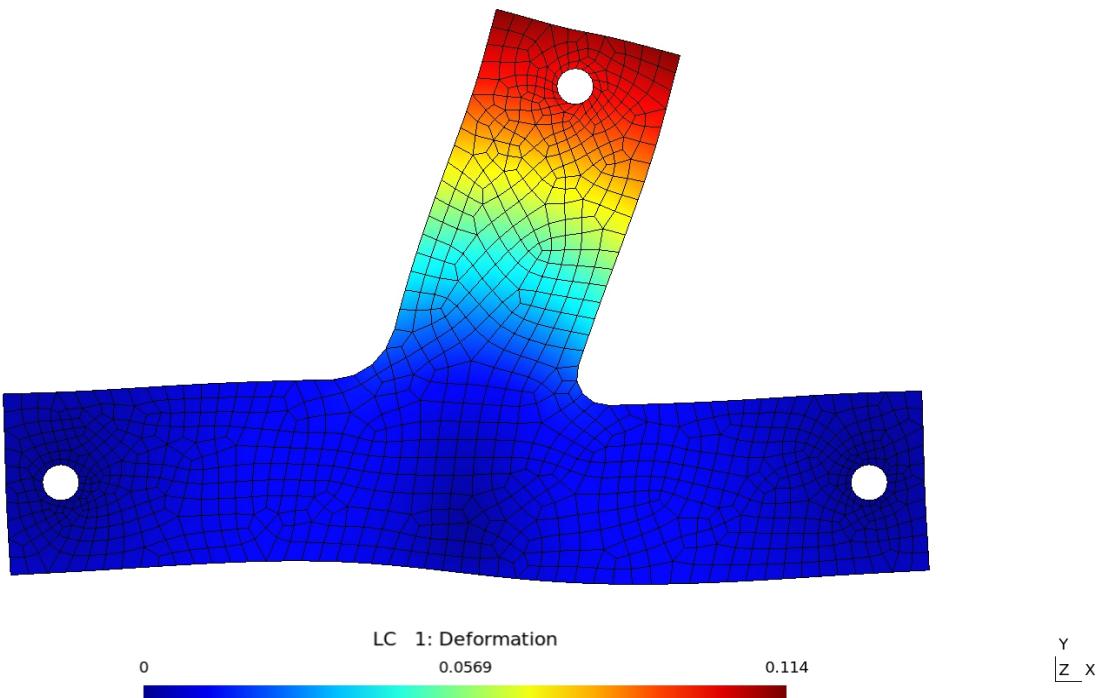


Figure 2.2-8: T-Junction, Deformation

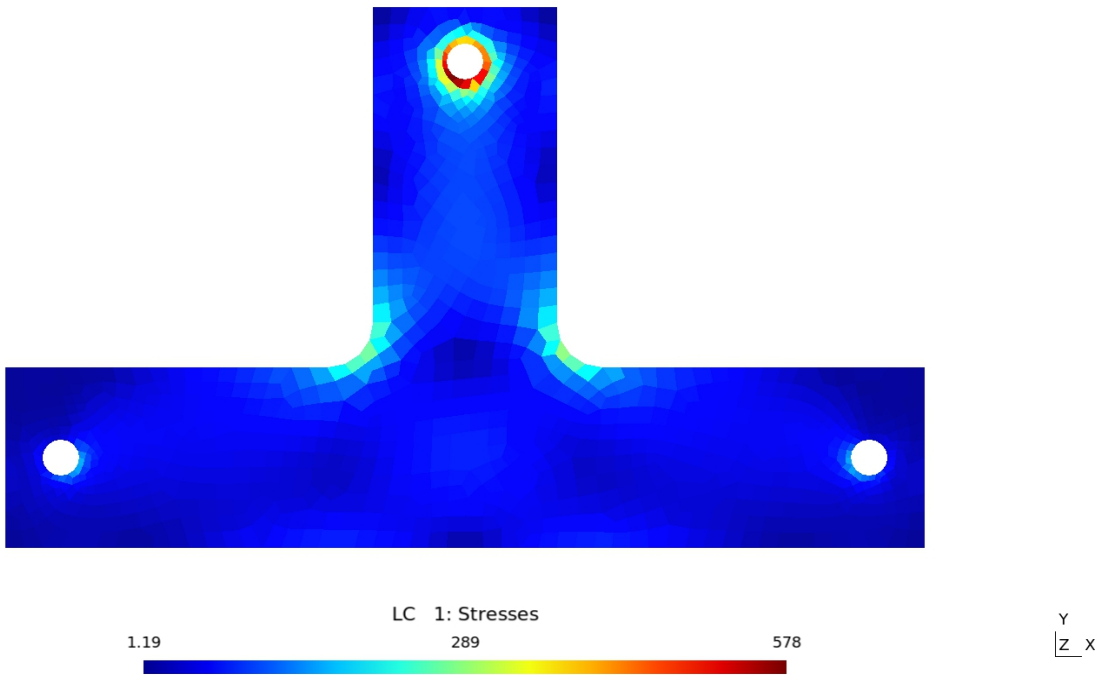


Figure 2.2-9: T-Junction, von Mises Stresses

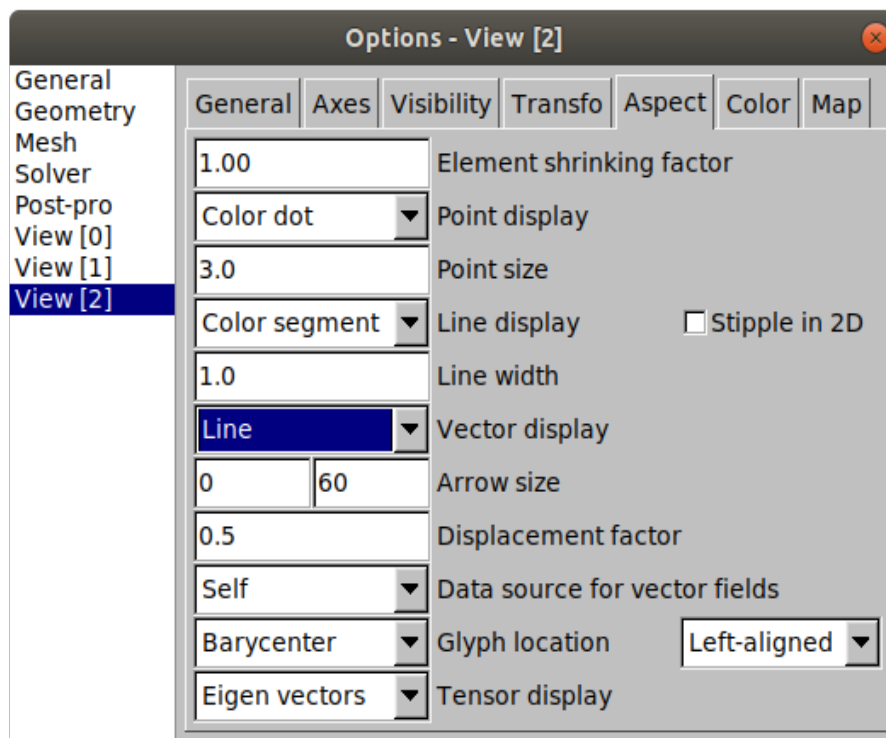


Figure 2.2-10: Gmsh: View Aspect Tab

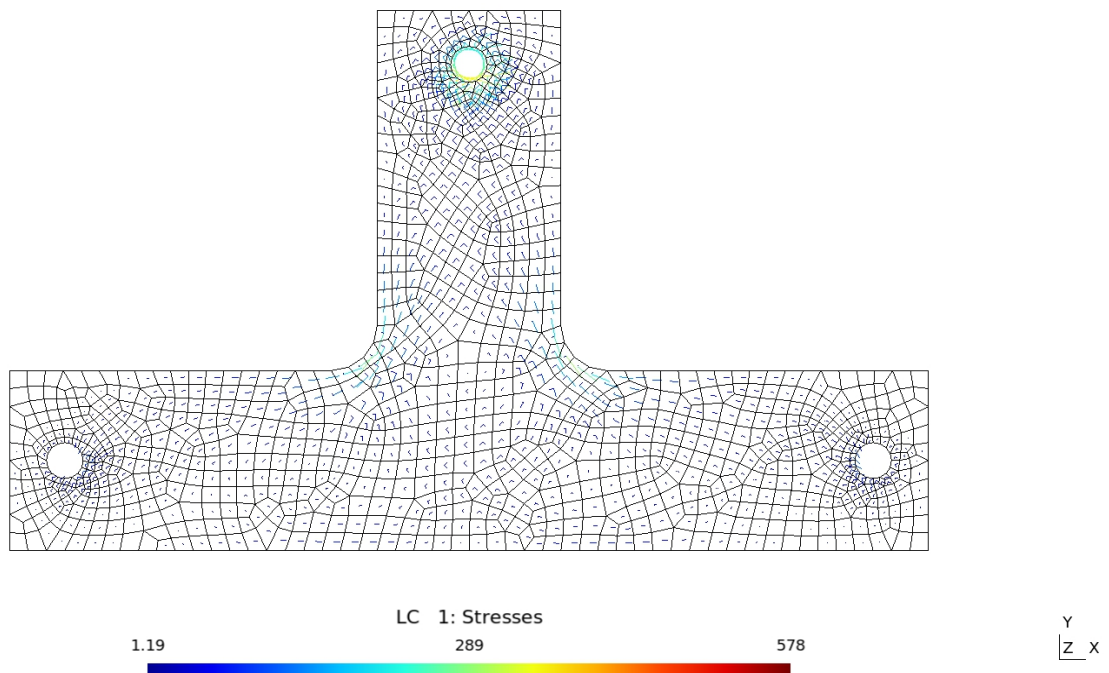


Figure 2.2-11: T-Junction, Principal Stress Trajectories

## 2.3 Membrane with a Hole

### Summary

Directory:	exa/solid/statresp/hole2d
Objectives:	<ul style="list-style-type: none"> <li>• learn how to define a structured 2-dimensional mesh using Gmsh</li> <li>• learn how to define node sets</li> <li>• learn how to use sets to restrict printed output</li> <li>• study the effect of using different element types and different mesh sizes</li> </ul>
Dimension:	2
Elements:	<b>q4, q8, q9, q4r</b>
Loads:	prescribed displacements
Functions:	<b>mfs_import, mfs_new, mfs_print, mfs_stiff, mfs_statresp, mfs_results, mfs_export</b>

### Problem Description

Compute the stresses in the membrane with hole shown in Figure 2.3-1 using a structured mesh. At the right end of the membrane, a displacement  $u_P$  in positive  $x$ -direction is prescribed. At the left end, the same displacement  $u_P$  in negative  $x$ -direction is prescribed.

Data:  $a = 100$  mm,  $b = 50$  mm,  $r = 20$  mm,  $R = 2$  mm, thickness  $t = 5$  mm, Young's

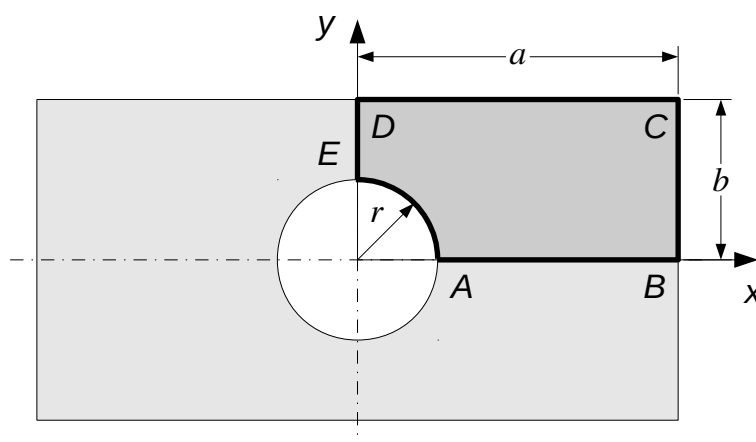


Figure 2.3-1: Membrane with Hole

modulus  $E = 210 \text{ GPa}$ , Poisson's ratio  $\nu = 0.3$ ,  $u_P = 0.1 \text{ mm}$

### Model Definition

Because the problem has two planes of symmetry, only one quarter of the structure need be modelled, see Figure 2.3-1. Along the edge  $BC$ , the displacement in  $x$ -direction is prescribed. Along the edges  $AB$  and  $DE$ , symmetry boundary conditions are applied, i.e. along the edge  $AB$ , the displacement in  $y$ -direction is zero, and along the edge  $DE$ , the displacement in  $x$ -direction is zero.

To get a structured mesh with quadrilateral elements only, the structure is subdivided into three quadrilateral surfaces, see Figure 2.3-2.

File `hole.geo` contains the commands to define the geometry and the mesh. In case of a structured mesh, the mesh density is defined by specifying the number of nodes along a line. The progression is the quotient between the lengths of two successive elements. E. g., a progression of 2 means that the length of each element in a series is twice the length of the preceding element. The progression is used for lines 1, 6 and 8 in order to get a finer mesh near the hole.

The definition of points, lines and surfaces is as explained in Example 2.2. Arcs are defined by specifying the start point, the centre and the end point. The arc is drawn clockwise from the start point to the end point.

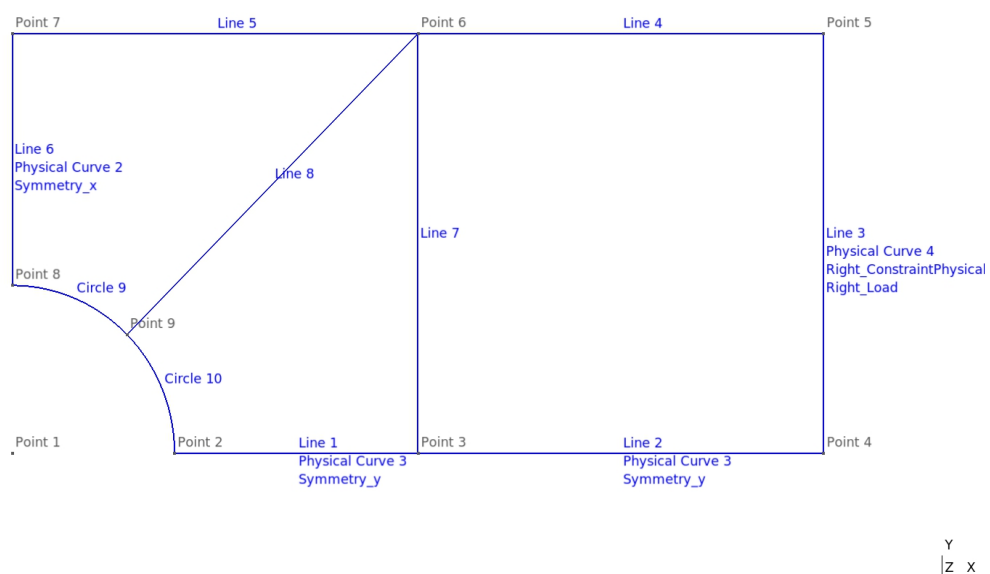


Figure 2.3-2: Hole, Geometry

```
// Dimensions [mm]

DefineConstant [ a = 100,
                 b = 50,
                 r = 20 ];

// Meshing parameters

np = GetValue("Number of Nodes along Line?", 11);
p  = GetValue("Progression?", 1.1);

Mesh.ElementOrder =
GetValue("Element Order: 1 = Linear, 2 = Quadratic?", 1);
If (Mesh.ElementOrder == 2)
    Mesh.SecondOrderIncomplete =
    GetValue("Incomplete Order: 0 = no, 1 = yes?", 0);
EndIf

Mesh.RecombineAll = 1;
Mesh.Smoothing = 1;

// Geometry

Point(1) = {0, 0, 0};
Point(2) = {r, 0, 0};
Point(3) = {b, 0, 0};
Point(4) = {a, 0, 0};
Point(5) = {a, b, 0};
Point(6) = {b, b, 0};
Point(7) = {0, b, 0};
Point(8) = {0, r, 0};
Point(9) = {r * Cos(Pi/4), r * Sin(Pi/4), 0};

Line(1) = {2, 3};
Line(2) = {3, 4};
Line(3) = {4, 5};
Line(4) = {5, 6};
Line(5) = {6, 7};
Line(6) = {8, 7};
Line(7) = {3, 6};
Line(8) = {9, 6};

Circle( 9) = {8, 1, 9};
Circle(10) = {9, 1, 2};

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

Plane Surface(1) = {1};
Plane Surface(2) = {2};
Plane Surface(3) = {3};

Physical Surface("Membrane") = {1, 2, 3};
```

```

Physical Line("Symmetry_x") = {6};
Physical Line("Symmetry_y") = {1, 2};
Physical Line("Right_Constraint") = {3};
Physical Line("Right_Load") = {3};

Physical Line("Hole") = {9, 10};

// Meshing commands

Transfinite Line{1, 6, 8} = np Using Progression p;
Transfinite Line{2 : 5, 7, 9, 10} = np;

Transfinite Surface{1 : 3};

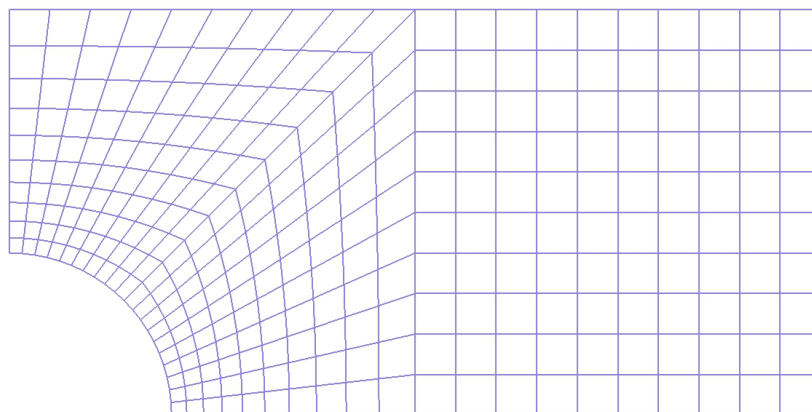
```

The physical line "**Hole**" will be used to define a set of nodal points that will be used to control printed output.

The last three commands define the structured mesh. The **Transfinite Line** command is used to specify the number of nodes to be created along the lines. The **Transfinite Surface** command defines the transfinite surface to be meshed to consist of all three geometric surfaces.

The mesh can be generated as described in Example 2.2. A typical mesh can be seen in Figure 2.3-3.

The translation data to generate a Mefisto model from file `hole.msh` can be found at the beginning of file `hole.m`. The definition of `opts.chkelt` is optional. If given a value of 0 no checks of the element data will be performed.



y  
z\_x

Figure 2.3-3: Hole, Typical Mesh



This is recommended only if you are sure that the definition of the elements is correct.

```
# Example 3: Membrane with hole
#
# -----

opts.chkelts = 1;

# Data (N, mm)

E = 210000;      % Youngs's modulus
ny = 0.3;        % Poisson ratio
t = 5;           % Thickness
u = 0.1;         % Prescribed displacement

name = {"3 = q4", "10 = q9", "16 = q8"};

# Output file

fid = fopen("hole.res", "wt");

# Translation data

HOLE = struct("type", "solid", "subtype", "2d");

geom = struct("t", t);
mat = struct("type", "iso", "E", E, "ny", ny);

HOLE.Membrane = struct("type", "elements",
                      "name", {name},
                      "geom", geom,
                      "mat", mat);

HOLE.Symmetry_x = struct("type", "constraints",
                        "name", "prescribed",
                        "dofs", 1);

HOLE.Symmetry_y = struct("type", "constraints",
                        "name", "prescribed",
                        "dofs", 2);

HOLE.Right_Constraint = struct("type", "constraints",
                              "name", "prescribed",
                              "dofs", 1);

HOLE.Right_Load = struct("type", "loads",
                        "name", "disp",
                        "data", u);

HOLE.Hole = struct("type", "nodeset");
```

The last line tells Mefisto to build a node set **Hole** containing the nodal points in physical group **"Hole"**.

## Analysis

The commands to run the analysis are essentially the same as in Example 2.2.

```
# Import msh file

model = mfs_import(fid, "hole.msh", "msh", HOLE);

# Create component

hole = mfs_new(fid, model, opts);
mfs_print(fid, hole, "load", "disp");

# Analysis

hole = mfs_stiff(hole);
hole = mfs_statresp(hole);
hole = mfs_results(hole, "statresp", "element");

# Print reaction loads and displacements of nodes in set Hole

mfs_print(fid, hole, "statresp", "reac");
mfs_print(fid, hole, "statresp", {"disp", "Hole"});

# Export results go Gmsh

mfs_export("hole.pos", "msh", hole,
           "statresp", "disp", "reac", "stress");

fclose(fid);
```

## Postprocessing

Figures 2.3-4 and 2.3-5 show the von Mises stresses and the principal stress trajectories. How to generate these pictures with Gmsh is described in detail in Example 2.2.

The results have been obtained with the proposed default values of the mesh parameters.

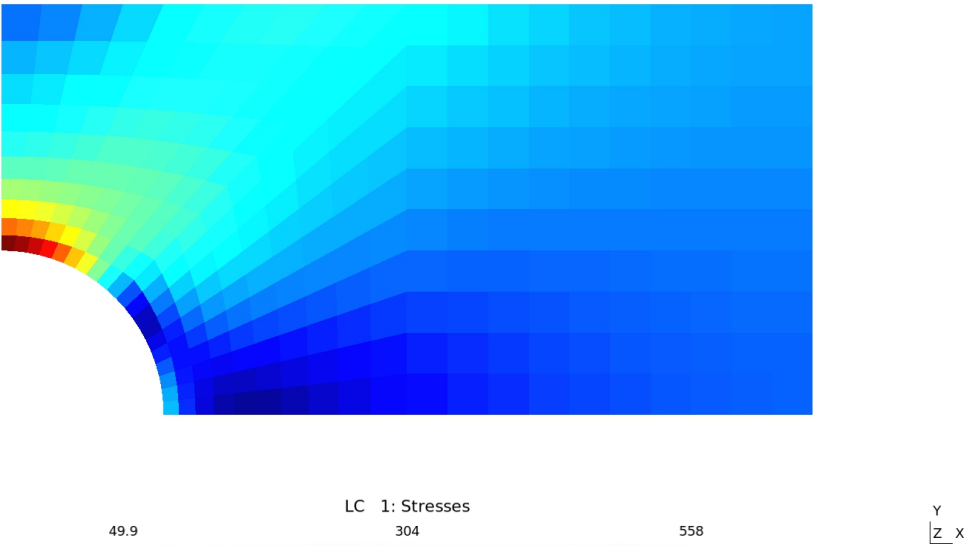


Figure 2.3-4: Hole, von Mises Stresses

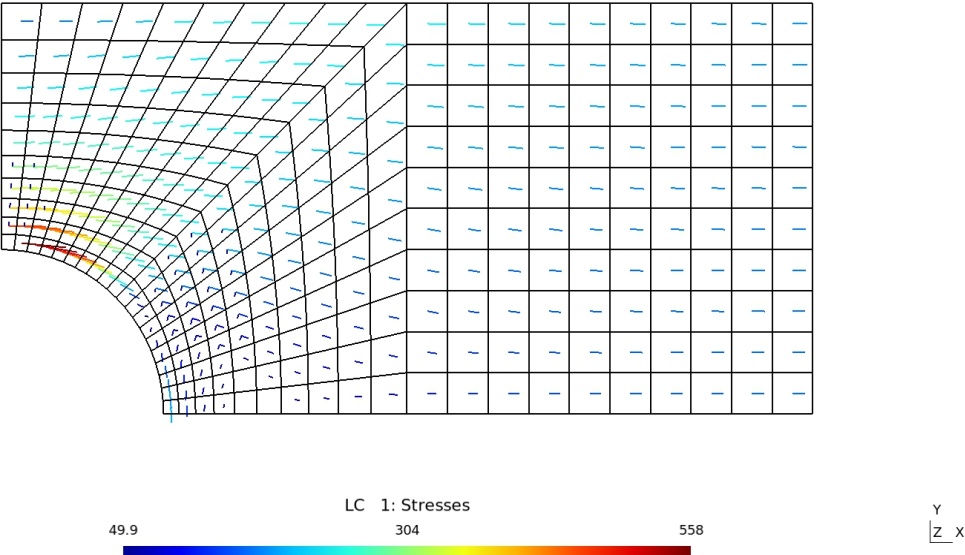


Figure 2.3-5: Hole, Principal Stress Trajectories

## 2.4 3-dimensional Truss

### Summary

Directory:	exa/solid/statresp/truss3d
Objectives:	<ul style="list-style-type: none"> <li>• learn how to define a simple 3-dimensional model</li> <li>• learn some additional postprocessing features of Gmsh</li> </ul>
Dimension:	3
Elements:	<b>r2</b>
Loads:	concentrated nodal point forces
Functions:	<b>mfs_new, mfs_print, mfs_stiff, mfs_statresp, mfs_results, mfs_export</b>

### Problem Description

Compute the deformation and the forces in all rods of the truss structure shown in Figure 2.4-1.

Data:  $a = 800$  mm,  $b = 600$  mm,  $G_y = 3$  kN,  $G_z = 6$  kN, cross section area  $A = 25$  mm<sup>2</sup>, Young's modulus  $E = 210$  GPa

### Model Definition

For a truss structure, there is no advantage to use Gmsh for the model generation. The definition of the Mefisto model is essentially the same as in Ex-

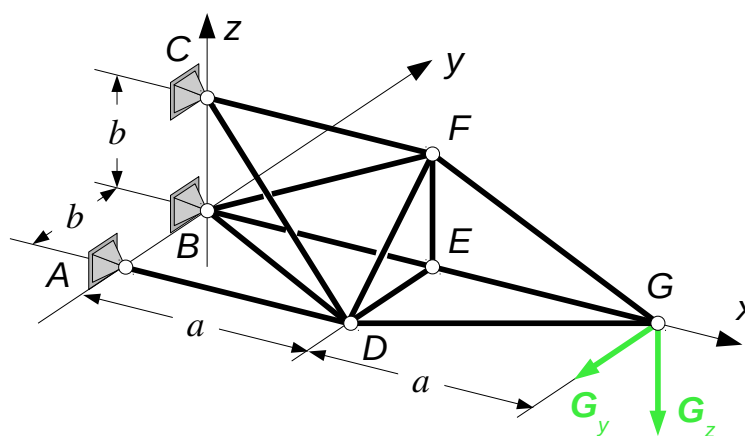


Figure 2.4-1: 3-dimensional Truss

ample 2.1. The only difference is the model subtype which is now "3d", and, of course, three coordinates must be defined for each nodal point.

```
# Example: 3-dimensional truss
#
# -----

# Data

a =      800;    % Length in mm
b =      600;    % Length in mm
Gy =    -3000;    % Force in N
Gz =    -6000;    % Force in N
E =     2.1E6;    % Young's modulus in MPa
A =       25;    % Cross section area in mm^2

# Output file

fid = fopen("truss.res", "wt");

# Model definition

model = struct("type", "solid", "subtype", "3d");

nodes(1).id = 1; nodes(1).coor = [      0  -b,  0]; % A
nodes(2).id = 2; nodes(2).coor = [      0,   0,  0]; % B
nodes(3).id = 3; nodes(3).coor = [      0,   0,  b]; % C
nodes(4).id = 4; nodes(4).coor = [      a,  -b,  0]; % D
nodes(5).id = 5; nodes(5).coor = [      a,   0,  0]; % E
nodes(6).id = 6; nodes(6).coor = [      a,   0,  b]; % F
nodes(7).id = 7; nodes(7).coor = [ 2 * a,   0,  0]; % G

model.nodes = nodes;

elem( 1).id =  1; elem( 1).nodes = [1, 4]; % AD
elem( 2).id =  2; elem( 2).nodes = [2, 4]; % BD
elem( 3).id =  3; elem( 3).nodes = [2, 5]; % BE
elem( 4).id =  4; elem( 4).nodes = [2, 6]; % BF
elem( 5).id =  5; elem( 5).nodes = [3, 4]; % CD
elem( 6).id =  6; elem( 6).nodes = [3, 6]; % CF
elem( 7).id =  7; elem( 7).nodes = [4, 5]; % DE
elem( 8).id =  8; elem( 8).nodes = [4, 6]; % DF
elem( 9).id =  9; elem( 9).nodes = [4, 7]; % DG
elem(10).id = 10; elem(10).nodes = [5, 6]; % EF
elem(11).id = 11; elem(11).nodes = [5, 7]; % EG
elem(12).id = 12; elem(12).nodes = [6, 7]; % FG

geom = struct("A", A);
mat = struct("type", "iso", "E", E);

for k = 1 : length(elem)
    elem(k).type = "r2";
    elem(k).geom = geom;
    elem(k).mat = mat;
end
```

```
model.elements = elem;  
  
hinge = struct("id", {1, 2, 3}, "dofs", 1 : 3);  
model.constraints.prescribed = hinge;  
  
force = struct("id", 7, "data", [0, Gy, Gz]);  
model.loads.point = force;
```

The resulting finite element model can be seen in Figure 2.4-2. To produce this plot, the font size has been increased in the Options-General Aspect tab (see Figure 2.4-3).

### Analysis

The Mefisto functions to perform the analysis are basically the same as in the previous examples. Thus, they are not repeated here.

### Postprocessing

In Gmsh, there is no difference between 2-dimensional and 3-dimensional structures. Thus, the steps needed to produce a plot of the deformed structure or a plot showing the forces in the rods are the same as those described in Example 2.1. Figure 2.4-4 shows the forces in the rods. Also for this plot, the font size has been increased.

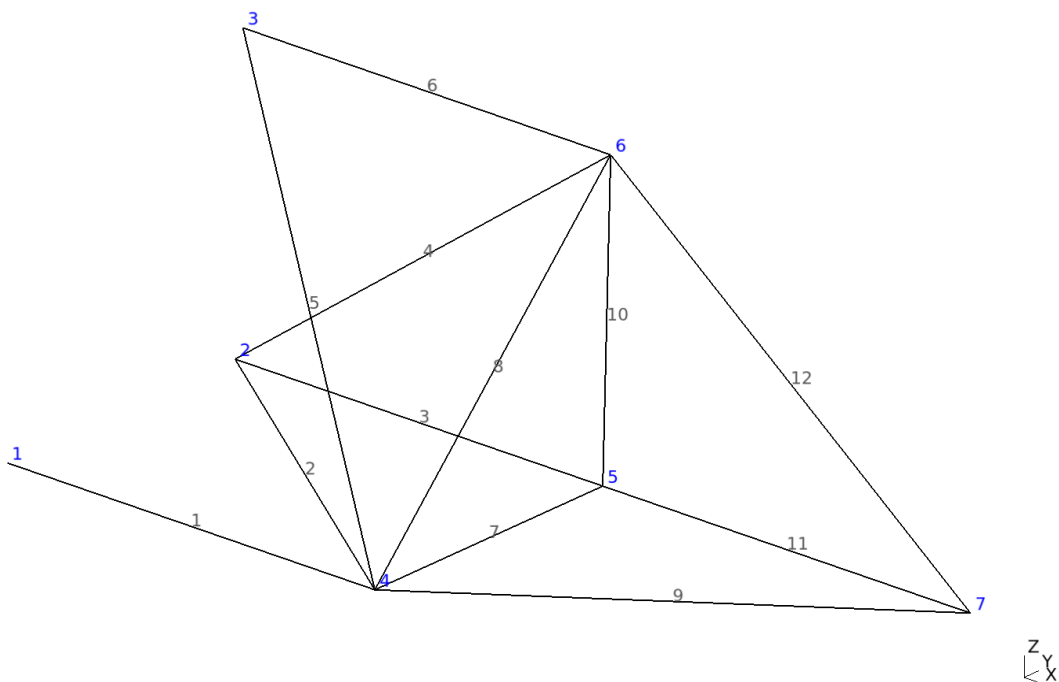


Figure 2.4-2: Truss, Finite Element Model

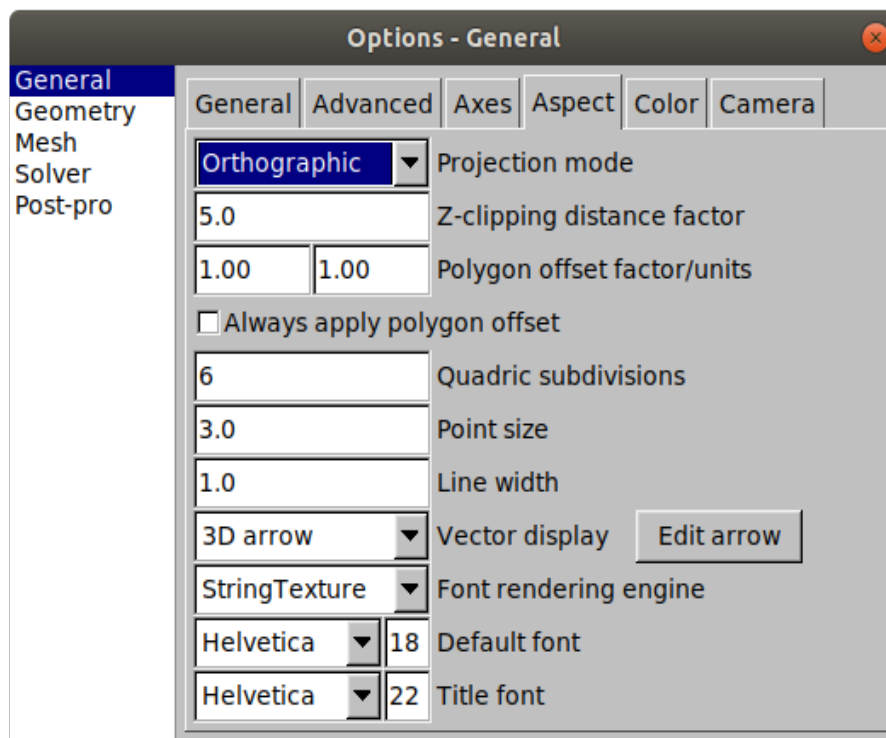


Figure 2.4-3: Gmsh: Options - General Aspect Tab

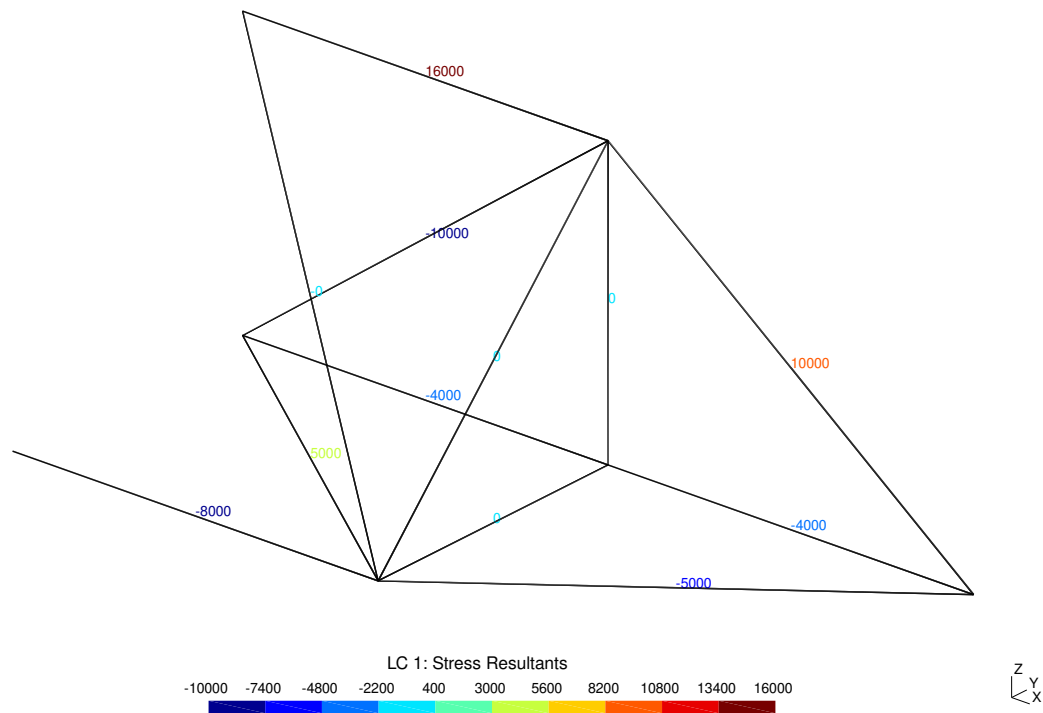


Figure 2.4-4: Truss, Forces in the Rods

## 2.5 3-dimensional Frame

### Summary

Directory:	exa/solid/statresp/frame3d
Objectives:	<ul style="list-style-type: none"> <li>• learn how to define 3-dimensional beam elements</li> <li>• learn how to visualize physical groups in Gmsh</li> <li>• learn how to check the definition of the beam axes using Gmsh</li> <li>• learn how to postprocess stress resultants using Gmsh</li> </ul>
Dimension:	3
Elements:	<b>b2</b>
Loads:	concentrated nodal point forces
Functions:	<b>mfs_beamsection, mfs_import, mfs_new, mfs_stiff, mfs_statresp, mfs_print, mfs_results, mfs_export</b>

### Problem Description

Compute the deformation and the stress resultants of the frame structure shown in Figures 2.5-1 to 2.5-4. The frame consists of a lower and an upper girder the cross section of which is a T-profile (see Figure 2.5-4). The girders are connected by diagonal bars with a rectangular cross section. The diagonal bars running from top left to bottom right are connected to the front side of the girders whereas the other diagonal bars are connected to the rear side. Thus, the structure is not symmetric with respect to the  $xz$ -plane.

The structure is loaded by a concentrated force  $F$  in negative  $z$ -direction applied at point  $P$ .

Data: Young's modulus  $E = 210$  GPa, Poisson's ratio  $\nu = 0.3$ ,  $F = 50$  kN

### Model Definition

Beam elements give exact results if loaded by concentrated loads. Thus, it would be sufficient to use only one element per segment. However, in deformed plots, a linear interpolation of the deformation is used. Thus, these plots look more realistic if a finer mesh is used.

Although, in this simple case, we could generate a mesh with variable element length using GNU Octave, we prefer to use Gmsh. In Gmsh, we just have to define points and lines and a typical element length. The identifiers of



the points and lines can be seen in Figure 2.5-5.

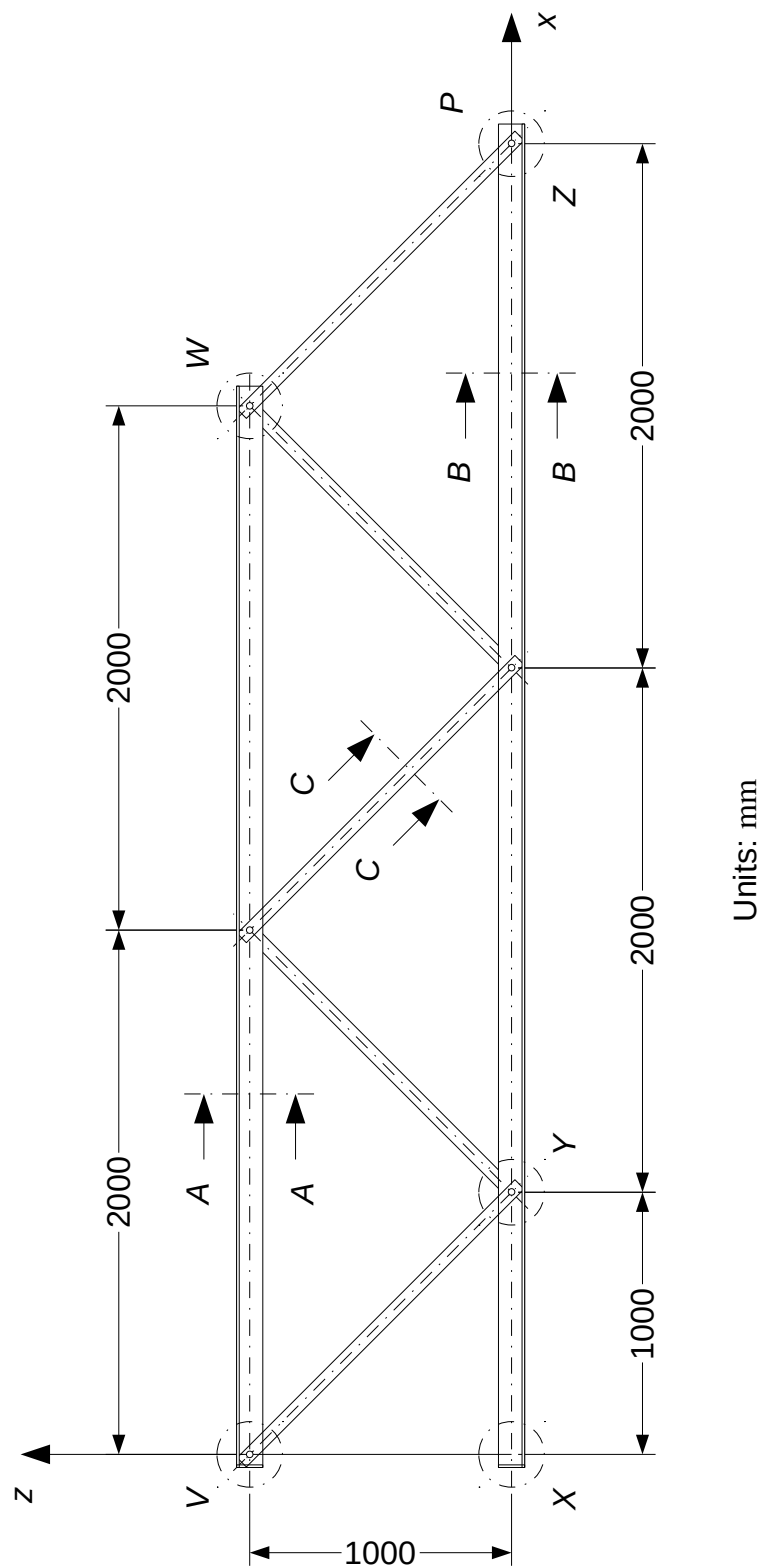


Figure 2.5-1: Draft of Frame Structure

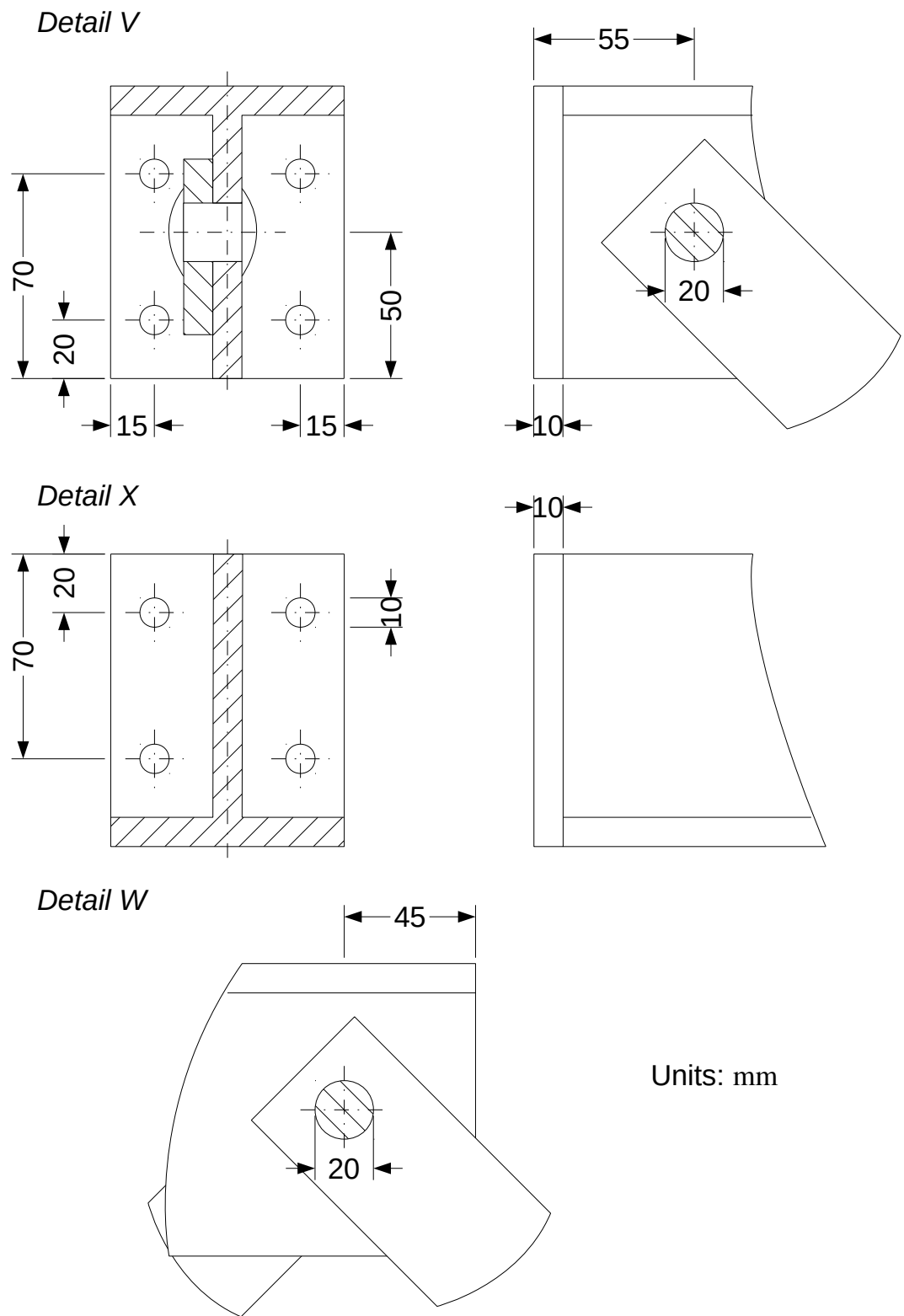
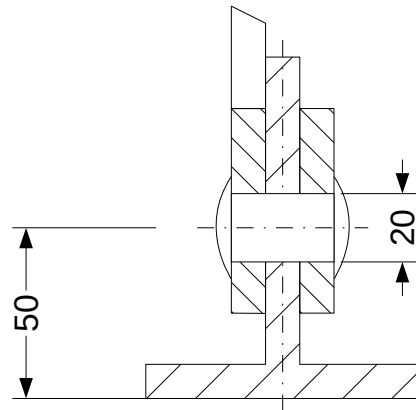
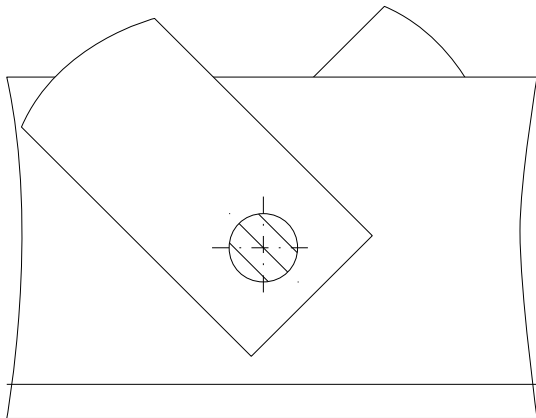
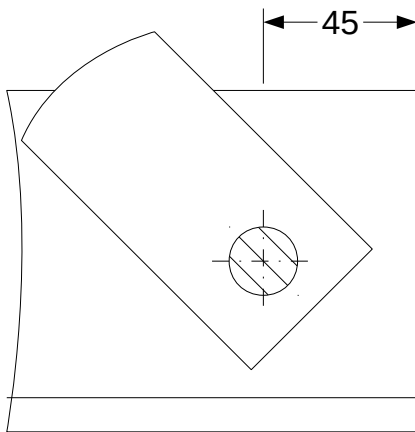


Figure 2.5-2: Draft of Frame Structure: Details Part 1

*Detail Y**Detail Z*

Units: mm

Figure 2.5-3: Draft of Frame Structure: Details Part 2

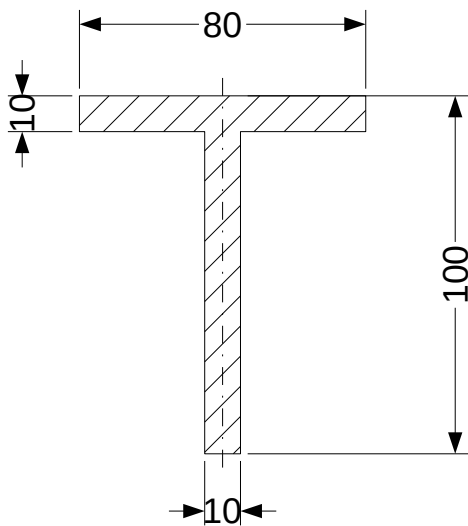
File `frame.geo` contains the following commands:

```
// Meshing parameters
lenb = GetValue("Typical Element Length?", 100);

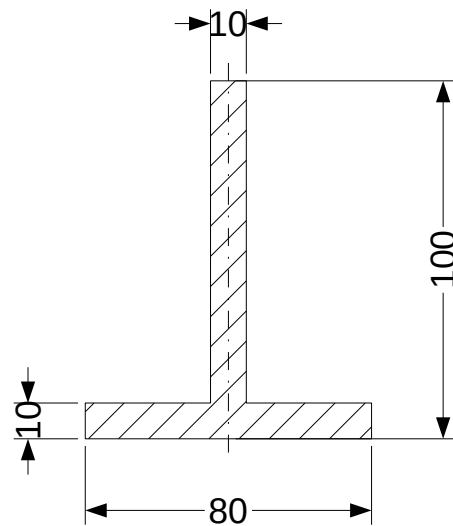
// Points
Point(1) = { 0, 0, 0, lenb};
Point(2) = { 1000, 0, 0, lenb};
Point(3) = { 3000, 0, 0, lenb};
Point(4) = { 5000, 0, 0, lenb};
Point(5) = { 0, 0, 1000, lenb};
Point(6) = { 2000, 0, 1000, lenb};
Point(7) = { 4000, 0, 1000, lenb};

// Lower Girder
Line(1) = {1, 2};
```

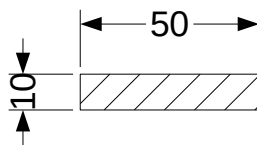
View A-A



View B-B



View C-C



Units: mm

Figure 2.5-4: Draft of Frame Structure: Views

```

Line(2) = {2, 3};
Line(3) = {3, 4};

Physical Line("Lower_Girder") = {1 : 3};

// Upper Girder

Line(4) = {5, 6};
Line(5) = {6, 7};

Physical Line("Upper_Girder") = {4, 5};

// Diagonal Bars

Line( 6) = {5, 2};
Line( 7) = {2, 6};
Line( 8) = {6, 3};
Line( 9) = {3, 7};
Line(10) = {7, 4};

```

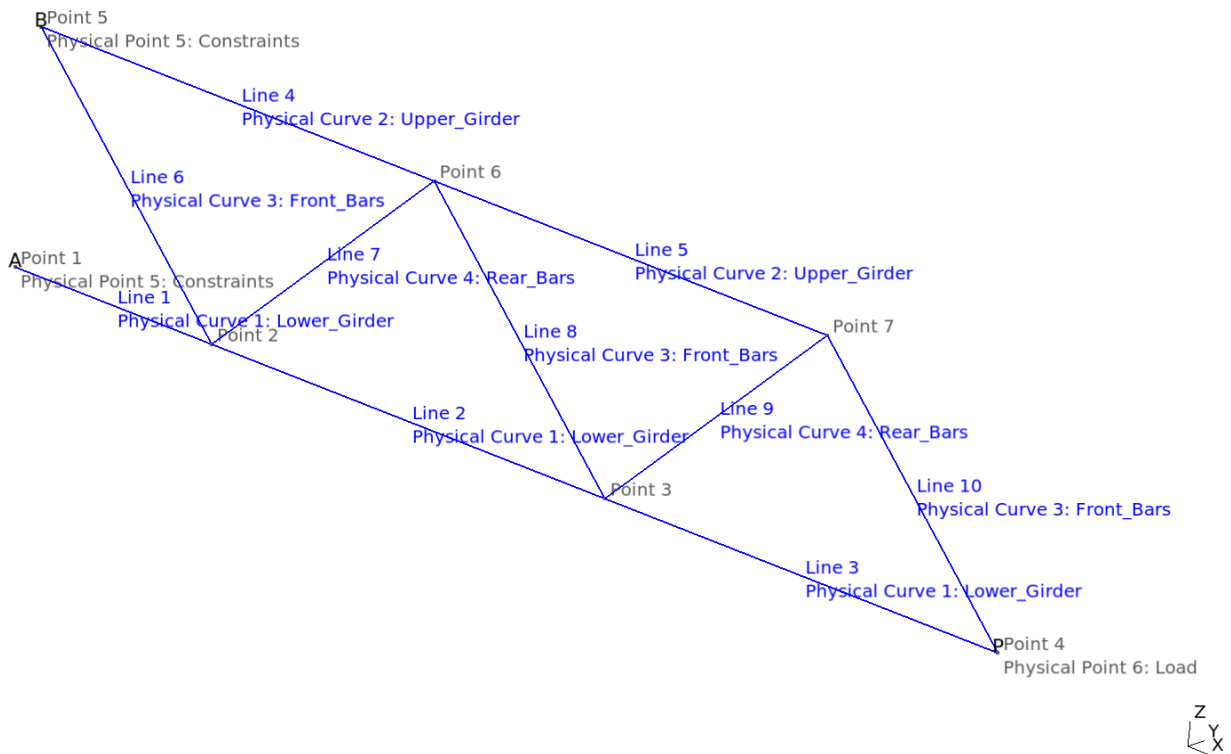


Figure 2.5-5: Frame, Geometry

```
Physical Line("Front_Bars") = {6, 8, 10};
Physical Line("Rear_Bars")  = {7, 9};

// Constraints

Physical Point("Constraints") = {1, 5};

// Load

Physical Point("Load") = {4};

// Annotations

fontsize = 22;
fonttype = 4;
textpos = 1;
font = fontsize + 2^8 * fonttype + 2^16 * textpos;

cA = Point{1};
cB = Point{5};
cP = Point{4};

View "Point Labels" {
  T3(cA[0], cA[1], cA[2], font){ "A" };
  T3(cB[0], cB[1], cB[2], font){ "B" };
  T3(cP[0], cP[1], cP[2], font){ "P" };
};
```

The beam elements are assigned to four different physical groups. There is one group for the beam elements of the lower girder, one group for the beam elements of the upper girder, one group for the elements of the front diagonal bars and one group for the rear diagonal bars. The beam elements of the lower and the upper girder have identical cross section properties, but the definition of the element z-axis is different. The same is true for the beam elements of the front and rear diagonal bars.

To check the physical groups, go to **Tools** → **Visibility** (see Figure 2.5-6). In the list browser that appears, you can select the physical groups you want to see (see Figure 2.5-7). You can use the Shift and the Ctrl keys to select more than one physical group.

The translation data are contained in file `frame.m`. The file begins with the definition of the data. Then, the output file is opened, and the model type and the model subtype as well as the material data are defined:

```
# Data (N, mm)

E = 210000; % Young's modulus
ny = 0.3; % Poisson's ratio
F = 50E3; % Force

# Output file

fid = fopen("frame.res", "wt");
```

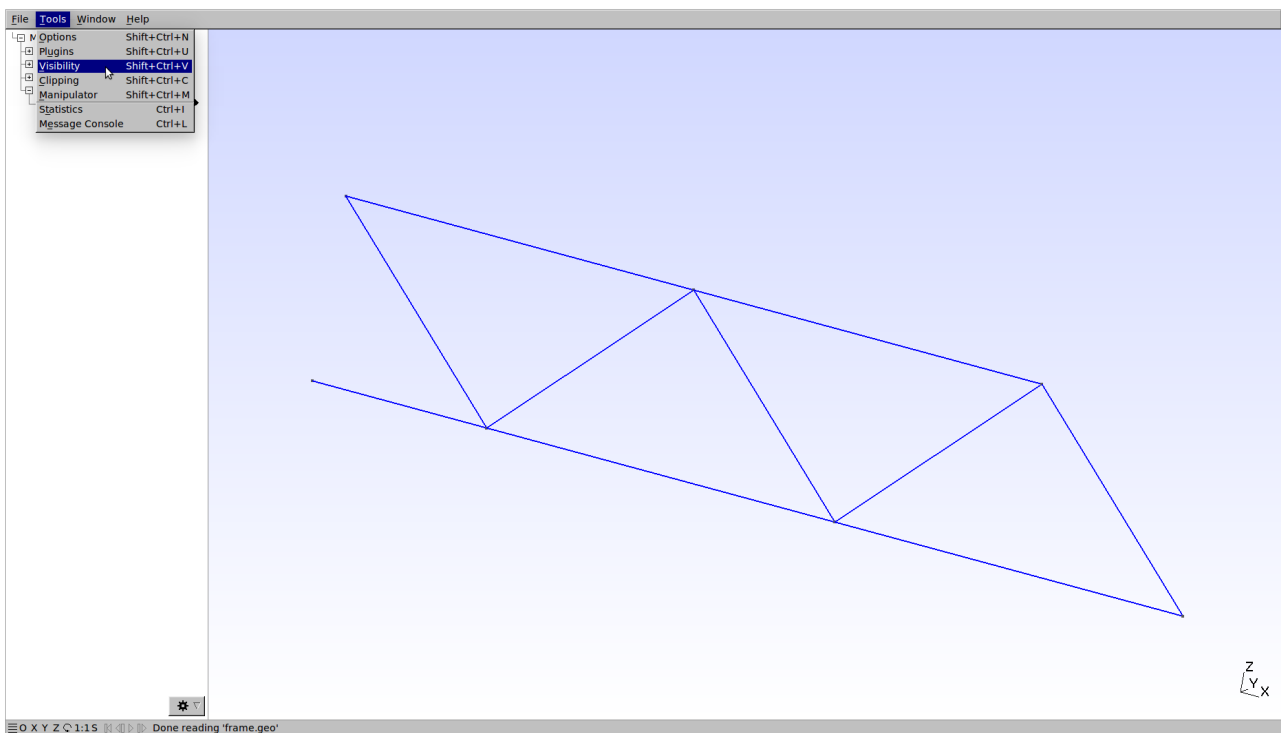


Figure 2.5-6: Gmsh: Tools → Visibility

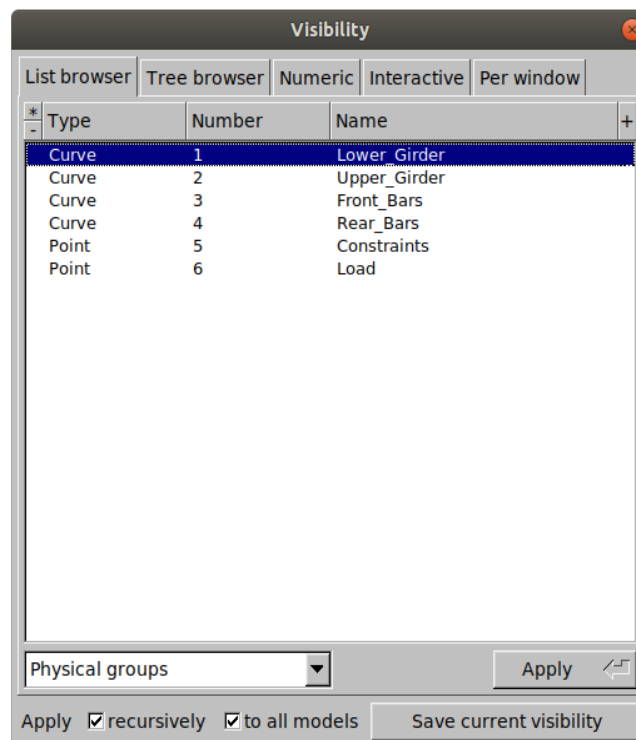


Figure 2.5-7: Gmsh: Visibility List Browser Tab

#### # Translation data

```
FRAME = struct("type", "solid", "subtype", "3d");
mat = struct("type", "iso", "E", E, "ny", ny);
```

Next, the cross section data of the beam elements are defined. Function **mfs\_beamsection** returns the cross section area, the moments of inertia, the torsional constant and the coordinates of the shear centre with respect to the element coordinate system. These data are returned in structure **geom**. In addition, the optional array **e** contains the values of  $e_y$  and  $e_z$  defining the location of the centre of area.

The elements are connected at the nodal points. By default, the nodal points of the beam elements coincide with the centre of area. In this case, however, the locations of the nodal points are different from the centre of area.

The element z-axis of the lower girder points upwards. The distance of the centre of area from the bottom of the cross section is  $e_y$ , but the distance of the nodal points where the diagonal bars are attached is 50 mm. Thus, the z-coordinate of the nodal point with respect to the element coordinate system is  $50 \text{ mm} - e_y$ . The same is true for the elements of the upper girder, with the element z-axis pointing downwards.

The element z-axis of the front diagonal bars points in positive y-direction.

The distance between the centre of area and the nodal point where the bar is connected to the girder is 10 mm. Thus, the z-coordinate of the nodal point with respect to the element coordinate system is 10 mm. The same is true for the elements of the rear diagonal bars, with the element z-axis pointing into negative y-direction.

Thus, the translation data for the beam elements are as follows:

```
[geom, e] = mfs_beamsection("T", 80, 100, 10);
geom.v     = [0, 0, 1];
geom.P     = [0, 50 - e(1)];

Lower_Girder = struct("type", "elements", "name", "b2",
                    "geom", geom, "mat", mat);

Upper_Girder = Lower_Girder;
Upper_Girder.geom.v = [0, 0, -1];

FRAME.Lower_Girder = Lower_Girder;
FRAME.Upper_Girder = Upper_Girder;

geom = mfs_beamsection("bar", 50, 10);
geom.v = [0, 1, 0];
geom.P = [0, 10];

Front_Bars = struct("type", "elements", "name", "b2",
                   "geom", geom, "mat", mat);

Rear_Bars = Front_Bars;
Rear_Bars.geom.v = [0, -1, 0];

FRAME.Front_Bars = Front_Bars;
FRAME.Rear_Bars = Rear_Bars;
```

The remaining translation data define the constraints and the load. At points A and B (see Figure 2.5-5) all three translations and all three rotations are set to zero. At point P, a concentrated force of 50 kN acting in negative z-direction is applied.

```
FRAME.Constraints = struct("type", "constraints",
                          "name", "prescribed",
                          "dofs", 1 : 6);

FRAME.Load = struct("type", "loads",
                   "name", "point",
                   "data", [0, 0, -F]);
```

## Analysis

The commands to run the analysis are essentially the same as in the previ-



ous examples. After the component has been created, the applied loads are written to the output file and the axes of the beam element coordinate systems are exported to file `axes.msh`. This file can be opened by Gmsh to check their correct definition.

```
# Import msh file

model = mfs_import(fid, "frame.msh", "msh", FRAME);

# Create component

frame = mfs_new(fid, model);
mfs_print(fid, frame, "load", "point");
mfs_export("axes.msh", "msh", frame, "mesh", "axes");

# Analysis

frame = mfs_stiff(frame);
frame = mfs_statresp(frame);
frame = mfs_results(frame, "statresp", "element");

mfs_print(fid, frame, "statresp", "reac");

# Export results to Gmsh

mfs_export("frame.pos", "msh", frame,
           "statresp", "disp", "reac", "resultant");

fclose(fid);
```

## Postprocessing

First, the correct definition of the beam axes is checked. To do so, open file `frame.msh` and merge file `axes.msh`. To see the element z-axis, deactivate all views but the one entitled “Element z-Axis”. Then, open the Options menu and go to the Aspect tab. Set Vector display to 3D arrow and Glyph location to Left-aligned (see Figure 2.5-8). Then, go to the Visibility tab, deactivate Show value scale and activate Draw element outlines (see Figure 2.5-9). The resulting Figure 2.5-10 shows that the definition of the element axes is correct.

Figure 2.5-11 shows the deformed frame structure. It can be seen that the lower girder, apart from bending in the xz-plane, is also subjected to bending in the xy-plane and to torsion. The torsion causes a bending of the diagonal bars. These effects are due to the asymmetric attachment of the diagonal bars to the girders.

Stress resultants can be visualized as explained in Example 2.1. In the View Visibility tab, you can select the resultant to be displayed. The correspond-

ence between fields and stress resultants can be seen in Table 2.5-1.

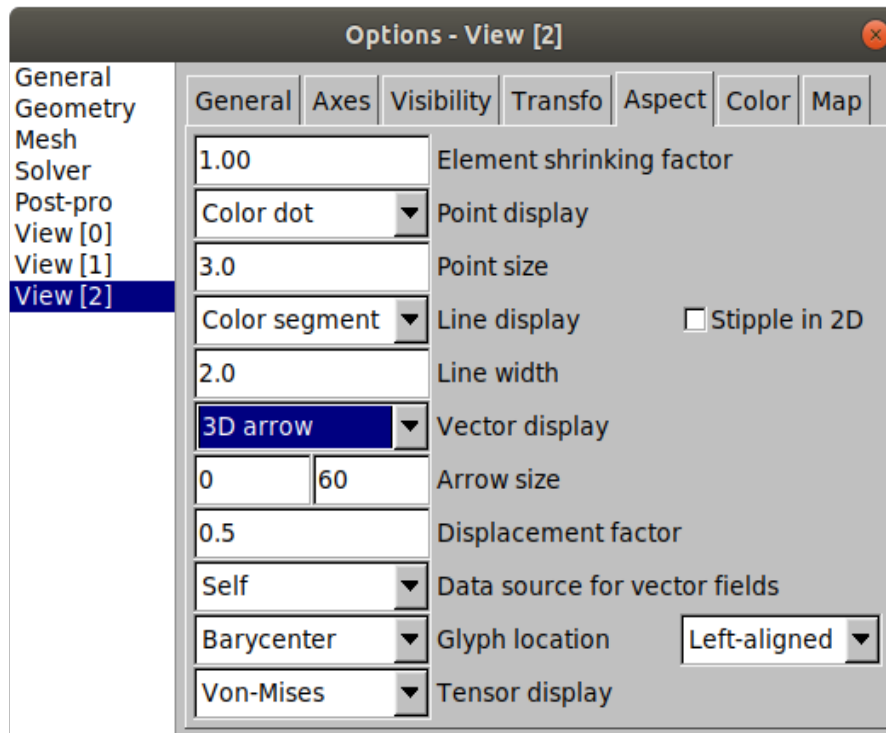


Figure 2.5-8: Gmsh: View Aspect Tab

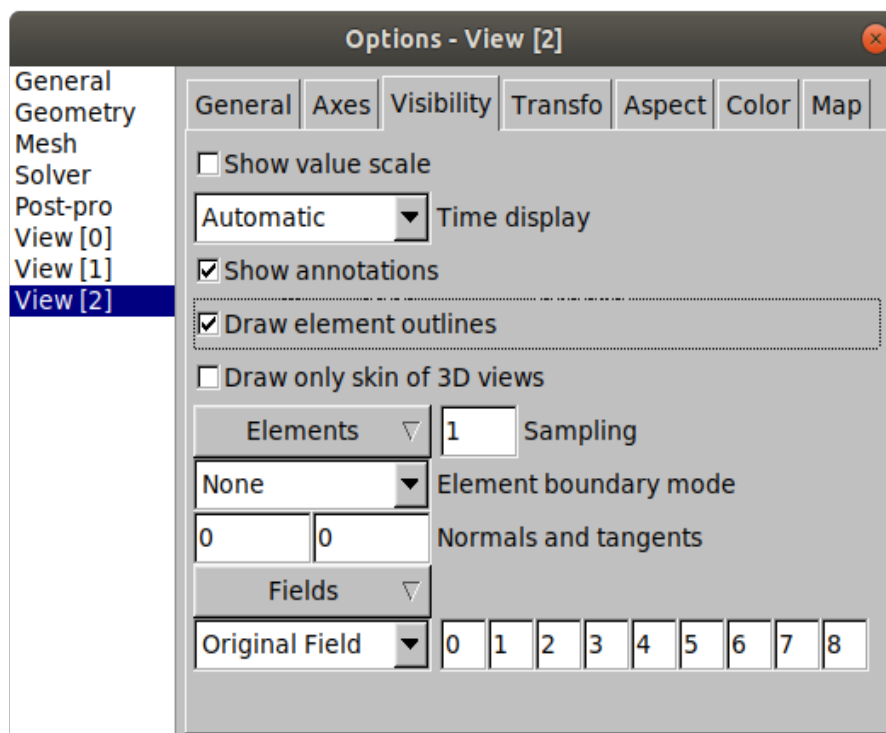


Figure 2.5-9: Gmsh: View Visibility Tab

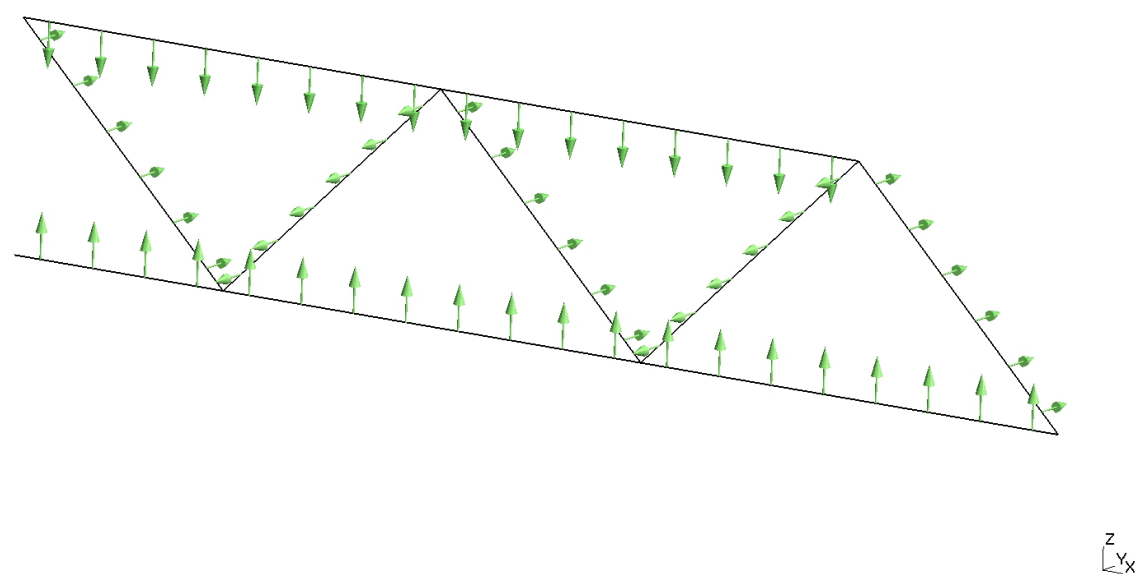


Figure 2.5-10: Frame, Beam Element z-Axes

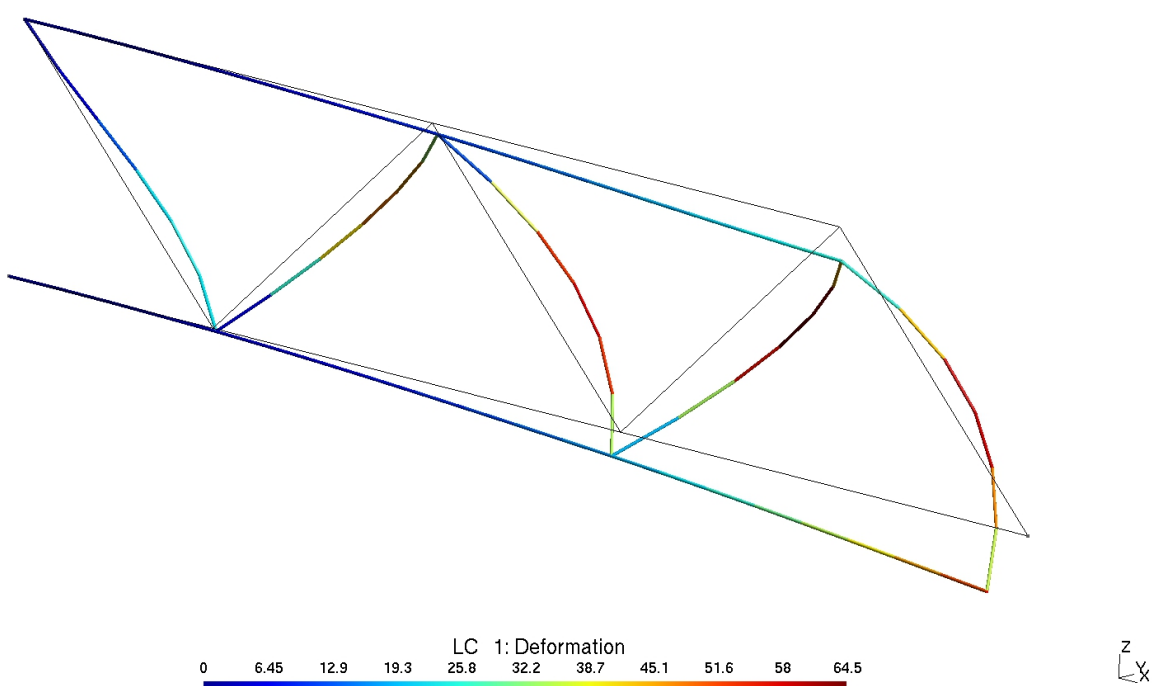
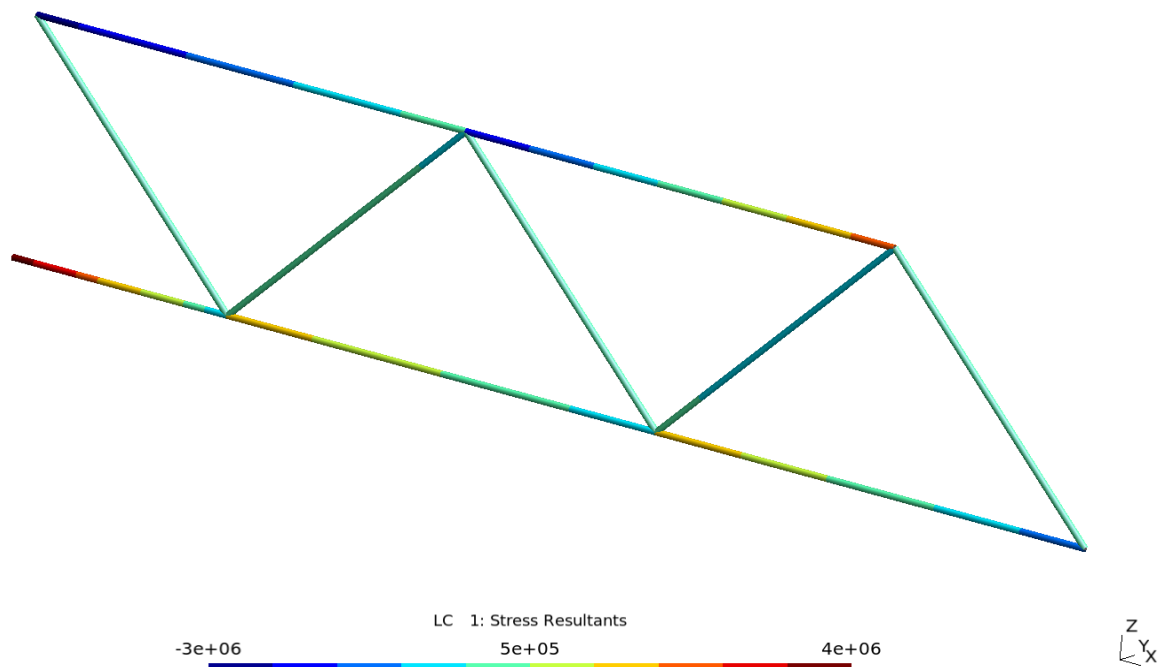


Figure 2.5-11: Frame, Deformed Structure

Figure 2.5-12: Frame, Bending Moment  $M_y$ 

Don't forget to adjust the range in the View General tab. As an example, Fig-

Stress resultant	$N$	$Q_y$	$Q_z$	$M_x$	$M_y$	$M_z$
Field	0	1	2	3	4	5

Tabelle 2.5-1: Fields and Stress Resultants

Figure 2.5-12 shows the bending moment about the element  $y$ -axis.

## 2.6 Torsion Box

### Summary

Directory:	exa/solid/statresp/tube3d
Objectives:	<ul style="list-style-type: none"> <li>• learn how to use the 3-dimensional membrane element</li> <li>• learn how to use graphic elements</li> <li>• learn how to define rigid body constraints</li> </ul>

Dimension:	3
Elements:	<b>q4</b> , <b>g2</b>
Constraints:	<b>rigbdy</b>
Loads:	prescribed displacements
Functions:	<b>mfs_linenodes</b> , <b>mfs_beamsection</b> , <b>mfs_new</b> , <b>mfs_export</b> , <b>mfs_stiff</b> , <b>mfs_statresp</b> , <b>mfs_results</b> , <b>mfs_print</b>

### Problem Description

Compute the stresses and the deformation of the thin-walled rectangular box shown in Figure 2.6-1. The load is a constant torsion moment which is applied by prescribing a rigid body rotation at the right end. Constraints are defined such that the box is free to warp. The results should agree with the theoretical results obtained from Bredt's theory.

Data:  $a = 100$  mm,  $t = 1$  mm, Young's modulus  $E = 210$  GPa, Poisson's ratio  $\nu = 0.3$ , twist angle  $\theta = 0.01$

### Model Definition

Quadrilateral membrane elements **q4** are used to model the tube. These elements have no stiffness perpendicular to the plane of the element. Thus, to each nodal point, at least two elements that are not coplanar have to be con-

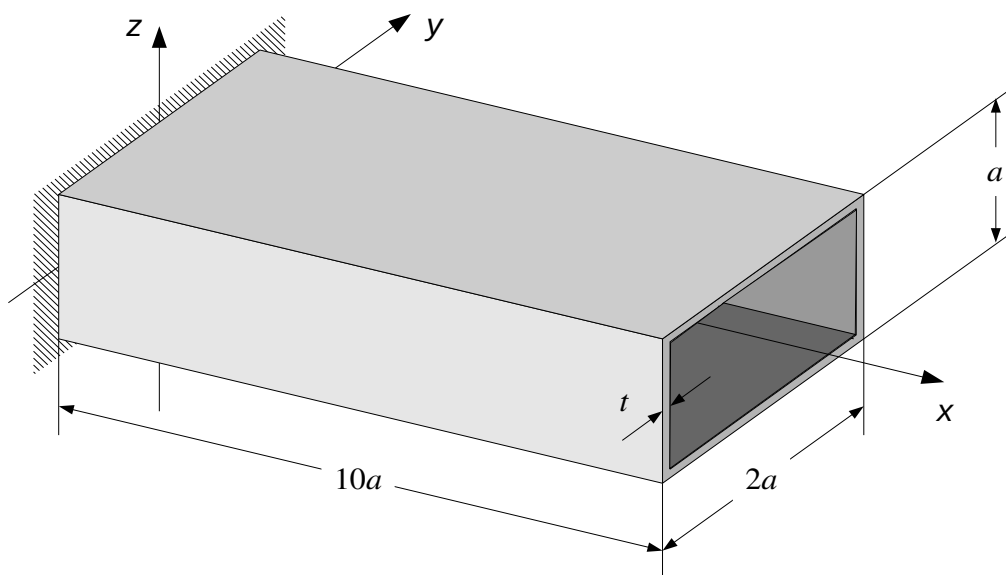


Figure 2.6-1: Torsion Box

nected. Hence, there can be nodal points only along the four edges of the tube. The length of the tube is subdivided into ten elements, with one row of elements in  $y$ -direction and one row of elements in  $z$ -direction.

At the left end of the tube, the displacements in  $y$ - and  $z$ -direction are constrained. On the right end, a linearized rotation about the  $x$ -axis is prescribed.

To prevent the rigid body translation in  $x$ -direction without constraining the warping, two horizontal rigid beams are added at the left end and two vertical rigid beams at the right end. The horizontal beams can rotate about the vertical axis  $AB$  and the vertical beams about the horizontal axis  $CD$ , see Figure 2.6-2.

To prescribe the rotation of the right end, the  $y$ - and  $z$ -displacements of the nodal points at this end are attached to a rigid body that has its origin in nodal point 2000, located in the middle between the four edge nodes. Then the rotation around the  $x$ -axis can be prescribed at nodal point 2000.

This simple model is defined in the GNU Octave script. The definition of the nodal points and of the elements is done as explained in the previous examples.

```
# Data

theta = 0.01; % Torsion angle (in rad)
a      = 100; % Length in mm
t      = 1;   % Thickness in mm
E      = 210000; % Young's modulus in MPa
ny     = 0.3; % Poisson's ratio
```

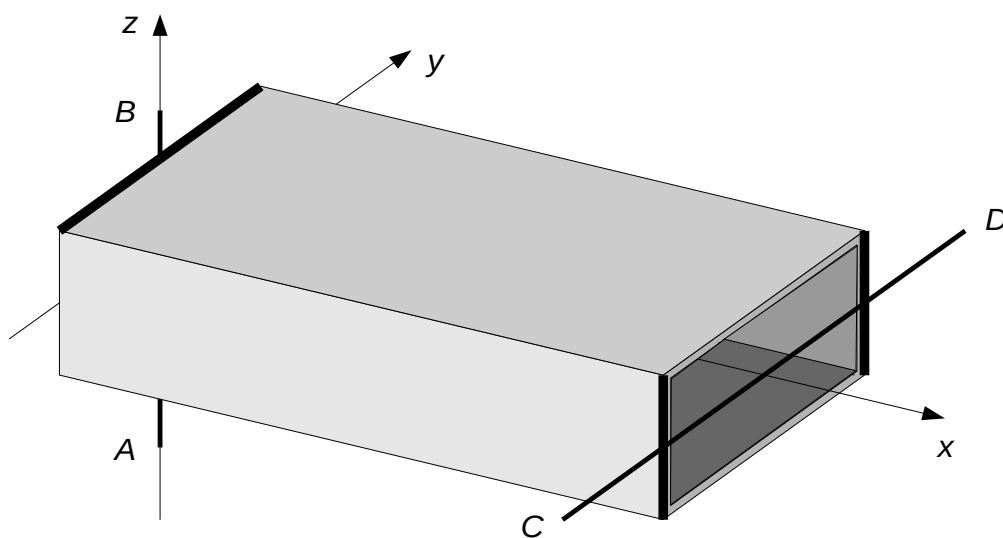


Figure 2.6-2: Box, Constraints

```

b      = 0.5 * a;

mat = struct("type", "iso", "E", E, "ny", ny);
geom = struct("t", t);

# Output file

fid = fopen("tube.res", "wt");

# Model definition

tube = struct("type", "solid", "subtype", "3d");

% Nodal points at left end

nodes( 1 : 4) = struct("id", {1, 2, 3, 4},
                      "coor", {[ 0, -a, -b], ...
                               [ 0,  a, -b], ...
                               [ 0,  a,  b], ...
                               [ 0, -a,  b]});

% Nodal points at right end

nodes( 5 : 8) = struct("id", {101, 102, 103, 104},
                      "coor", {[ 10 * a, -a, -b], ...
                               [ 10 * a,  a, -b], ...
                               [ 10 * a,  a,  b], ...
                               [ 10 * a, -a,  b]});

% Nodal points for rigid bodies at left end

nodes( 9 : 10) = struct("id", {1001, 1002},
                      "coor", {[ 0,  0, -b], ...
                               [ 0,  0,  b]});

% Nodal points for rigid bodies at right end

nodes(11 : 13) = struct("id", {2000, 2001, 2002},
                      "coor", {[ 10 * a,  0,  0], ...
                               [ 10 * a, -a,  0], ...
                               [ 10 * a,  a,  0]});

% Remaining nodal points

nodes = mfs_linenodes(nodes, 1, 101, 11 : 10 : 91);
nodes = mfs_linenodes(nodes, 2, 102, 12 : 10 : 92);
nodes = mfs_linenodes(nodes, 3, 103, 13 : 10 : 93);
nodes = mfs_linenodes(nodes, 4, 104, 14 : 10 : 94);

tube.nodes = nodes;

% Membrane elements

ix = 1 : 10;
for n = 1 : 4
    eltid = [n : 10 : 90 + n]';

```

```

id4    = 5 - n;
id1    = mod(id4, 4) + 1;
nodids = [id1 : 10 : 90 + id1;
          id1 + 10 : 10 : 100 + id1;
          id4 + 10 : 10 : 100 + id4;
          id4 : 10 : 90 + id4]';
quad(ix) = struct("id", num2cell(eltids),
                  "nodes", mat2cell(nodids, ones(10, 1)),
                  "type", "q4", "geom", geom,
                  "mat", mat);

ix += 10;
endfor

```

The finite element model created by these commands can be seen in Figure 2.6-3.

The following commands define the graphic elements. These elements are used for visualization only. They require neither geometry data nor material data.

```

% Graphic elements

graphic(1 : 4) = struct("id", {1001, 1002, 1003, 1004},
                        "nodes", {[1, 1001], [1001, 2], ...
                                [3, 1002], [1002, 4]},
                        "type", "g2", "geom", [], "mat", []);
graphic(5 : 8) = struct("id", {2001, 2002, 2003, 2004},
                        "nodes", {[101, 2001], [2001, 104], ...
                                [102, 2002], [2002, 103]},
                        "type", "g2", "geom", [], "mat", []);
graphic(9 : 12) = struct("id", {3001, 3002, 3003, 3004},

```

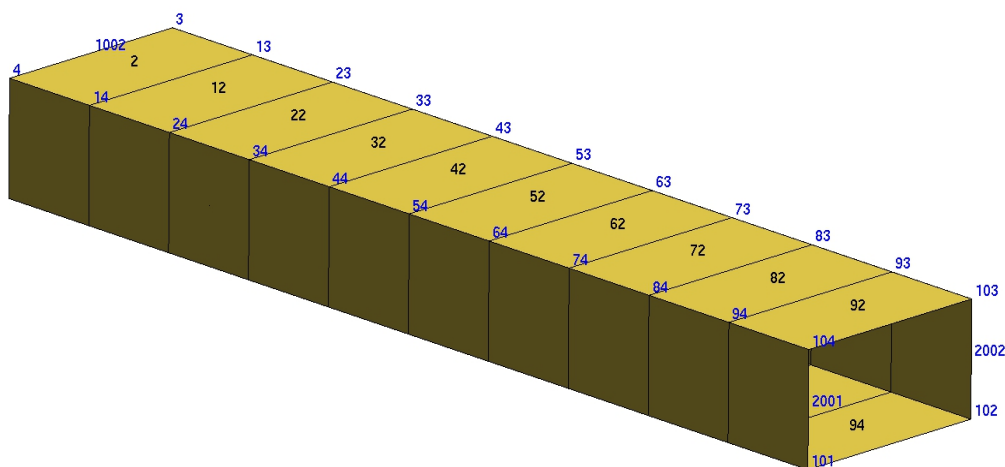


Figure 2.6-3: Box, Finite Element Model



```

        "nodes", {[2000, 101], [2000, 102], ...
                  [2000, 103], [2000, 104]}},
        "type", "g2", "geom", [], "mat", []);

tube.elements = [quad, graphic];

```

Next, we define the rigid bodies. Field **noda** defines the nodal point with the independent degrees of freedom. Independent degrees of freedom involved in a linear constraint are also called autonomous degrees of freedom. Field **nodd** defines the nodal points with dependent degrees of freedom, and field **dofs** defines the dependent degrees of freedom.

The first two rigid bodies are the horizontal rigid beams at the left end. The next two rigid bodies are the vertical rigid beams at the right end. The last rigid body is used to apply the prescribed motion.

```

% Rigid bodies

rigbdy(1) = struct("noda", 1001, "nodd", [ 1, 2], "dofs", 1);
rigbdy(2) = struct("noda", 1002, "nodd", [ 3, 4], "dofs", 1);
rigbdy(3) = struct("noda", 2001, "nodd", [101, 104], "dofs", 1);
rigbdy(4) = struct("noda", 2002, "nodd", [102, 103], "dofs", 1);
rigbdy(5) = struct("noda", 2000, "nodd", 101 : 104,
                  "dofs", [2, 3]);

tube.constraints.rigbdy = rigbdy;

```

Finally, we define the constraints and the prescribed motion.

```

% Supports

supports(1 : 4) = struct("id", {1, 2, 3, 4}, "dofs", [2, 3]);
supports(5 : 6) = struct("id", {1001, 1002}, "dofs", 1 : 5);
supports(7)      = struct("id", 2000, "dofs", 1 : 6);
supports(8 : 9) = struct("id", {2001, 2002},
                        "dofs", [1 : 4, 6]);

tube.constraints.prescribed = supports;

% Prescribed rotation

tube.loads.disp = struct("id", 2000,
                        "data", theta * [0, 0, 0, 1]);

```

## Analysis

The commands to run the analysis are basically the same as those in the previous examples. The item **ics1** requests the internal constraint loads to be

printed. Internal constraint loads are loads due to the linear constraints. They occur at the dependent and autonomous degrees of freedom.

#### # Analysis

```
cmp = mfs_new(fid, tube);
mfs_export("tube.msh", "msh", cmp, "mesh");

cmp = mfs_stiff(cmp);
cmp = mfs_statresp(cmp);
cmp = mfs_results(cmp, "statresp", "element");
mfs_print(fid, cmp, "statresp",
          "reac", "icsl", "stress");
mfs_export("tube.pos", "msh", cmp, "statresp", "disp");
```

Following the finite element analysis also the analytical solution is computed. First, function **mfs\_beamsection** is used to get the torsional constant

$$I_T = \frac{4 A_m^2}{\oint ds/t} = \frac{4 (2a^2)^2 t}{6a} = \frac{8}{3} a^3 t \quad .$$

Next, the shear modulus is obtained from

$$G = \frac{E}{2(1+\nu)} \quad .$$

The twist angle is related to the torque  $M_x$  by

$$\theta = \frac{10 a M_x}{G I_T}$$

yielding

$$M_x = \frac{G I_T \theta}{10 a} \quad .$$

The shear stress is given by

$$\tau_{sx} = \frac{M_x}{2 A_m t} = \frac{M_x}{4 a^2 t} \quad .$$

Finally, the displacements in x-direction at the edge points of the rectangle are obtained from

$$u_x = \pm \frac{M_x}{2 A_m G} \left( \frac{a}{t} - \frac{1}{8} \cdot 6 \frac{a}{t} \right) = \mp \frac{1}{16} \frac{M_x}{a t G} \quad .$$

#### # Analytical solution

```
geoma = mfs_beamsection("box", "thin", 2 * a, a, t);
G      = E / (2 * (1 + ny));
Mx     = G * geoma.IT * theta / (10 * a);
```

```

tsx = Mx / (4 * a^2 * t);
u   = Mx / (16 * a * t * G);

fprintf(fid, "\nAnalytical Solution:\n");
fprintf(fid, "    Mx = %10.3e, tauxy = %10.3e, ", Mx, tsx);
fprintf(fid, "u = %10.3e\n", u);

fclose(fid);

```

## Postprocessing

The result file shows that the torsional moment, the stresses and the displacements agree with the theoretical results. Reaction loads and internal constraint loads are both in equilibrium.

Mefisto 2.7: Building new component from input "tube"

```

Model Type = solid, Model Subtype = 3d

Number of nodes      = 49, Number of elements = 52
Number of element types = 2
Number of global degrees of freedom = 294
Number of local degrees of freedom = 112
Number of prescribed degrees of freedom = 34
Number of dependent degrees of freedom = 16

Number of load cases = 1

```

Component "cmp"

Reaction loads of loadcase 1

node	Fx	Fy	Fz	Mx	My	Mz
1	0.000e+00	-5.385e+03	2.692e+03	0.000e+00	0.000e+00	0.000e+00
2	0.000e+00	-5.385e+03	-2.692e+03	0.000e+00	0.000e+00	0.000e+00
3	0.000e+00	5.385e+03	-2.692e+03	0.000e+00	0.000e+00	0.000e+00
4	0.000e+00	5.385e+03	2.692e+03	0.000e+00	0.000e+00	0.000e+00
1001	1.000e-11	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
1002	-2.740e-11	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
2000	0.000e+00	0.000e+00	-5.821e-11	2.154e+06	0.000e+00	0.000e+00
2001	-3.456e-11	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
2002	-1.319e-11	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
Res.	-6.514e-11	9.095e-13	-5.321e-11	1.444e-08	5.634e-08	-2.137e-09

Internal constraint loads of loadcase 1

node	Fx	Fy	Fz	Mx	My	Mz
1	3.638e-12	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
2	6.366e-12	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
3	-1.432e-11	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
4	-1.307e-11	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
101	-5.457e-12	5.385e+03	-2.692e+03	0.000e+00	0.000e+00	0.000e+00
102	8.185e-12	5.385e+03	2.692e+03	0.000e+00	0.000e+00	0.000e+00
103	-2.137e-11	-5.385e+03	2.692e+03	0.000e+00	0.000e+00	0.000e+00
104	-2.910e-11	-5.385e+03	-2.692e+03	0.000e+00	0.000e+00	0.000e+00
1001	-1.000e-11	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
1002	2.740e-11	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
2000	0.000e+00	0.000e+00	5.821e-11	-2.154e+06	0.000e+00	0.000e+00
2001	3.456e-11	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
2002	1.319e-11	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00

Res. 0.000e+00 0.000e+00 0.000e+00 0.000e+00 -1.855e-09 -6.567e-10

#### Displacements of loadcase 1

node	ux	uy	uz	rx	ry	rz
1	-1.667e-02	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
2	1.667e-02	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
3	-1.667e-02	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
4	1.667e-02	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
11	-1.667e-02	5.000e-02	-1.000e-01	0.000e+00	0.000e+00	0.000e+00
...	...	...	...	...	...	...
104	1.667e-02	-5.000e-01	-1.000e+00	0.000e+00	0.000e+00	0.000e+00
1001	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00	-1.667e-04
1002	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00	1.667e-04
2000	0.000e+00	0.000e+00	0.000e+00	1.000e-02	0.000e+00	0.000e+00
2001	0.000e+00	0.000e+00	0.000e+00	0.000e+00	3.333e-04	0.000e+00
2002	0.000e+00	0.000e+00	0.000e+00	0.000e+00	-3.333e-04	0.000e+00

#### Stresses of loadcase 1

##### Membrane elements:

element	point	x	y	z	sigx	sigy	tauxy
1	1	50.00	-100.00	0.00	3.440e-14	1.032e-14	-5.385e+01
2	1	50.00	0.00	50.00	9.633e-14	1.751e-14	-5.385e+01
3	1	50.00	100.00	0.00	2.189e-14	6.568e-15	-5.385e+01
4	1	50.00	0.00	-50.00	-4.191e-14	-6.881e-15	-5.385e+01
11	1	150.00	-100.00	0.00	3.878e-14	3.440e-14	-5.385e+01
...	...	...	...	...	...	...	...
93	1	950.00	100.00	0.00	-7.819e-14	-2.346e-14	-5.385e+01
94	1	950.00	0.00	-50.00	-7.819e-14	-2.346e-14	-5.385e+01

#### Analytical Solution:

Mx = 2.154e+06, tauxy = 5.385e+01, u = 1.667e-02

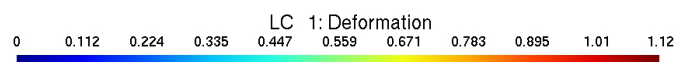
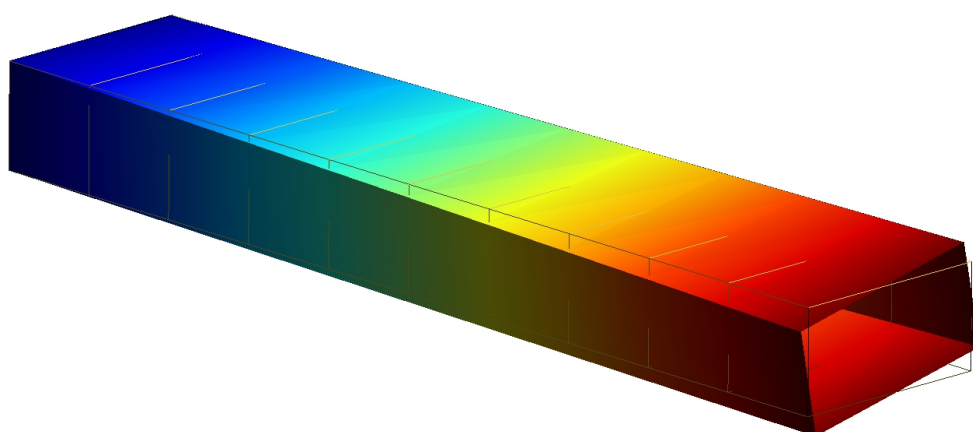


Figure 2.6-4: Box, Deformed Structure

Figure 2.6-4 shows a linear increase of the twist angle with the  $x$ -coordinate, as predicted by theory.

## 2.7 Scordelis-Lo Roof

### Summary

Directory:	exa/solid/statresp/roof
Objectives:	<ul style="list-style-type: none"> <li>• get familiar with shell elements</li> <li>• learn how to apply inertia loads</li> <li>• learn how to access nodal point and element results</li> <li>• learn how to work with element sets</li> </ul>
Dimension:	3
Elements:	<b>s4</b> , <b>s3</b>
Loads:	inertia loads
Functions:	<b>mfs_new</b> , <b>mfs_export</b> , <b>mfs_stiff</b> , <b>mfs_mass</b> , <b>mfs_statresp</b> , <b>mfs_getresp</b> , <b>mfs_results</b> , <b>mfs_print</b>

### Problem Description

Compute the displacements, stresses and stress resultants of the cylindrical shell roof shown in Figure 2.7-1. Use both **s4** and **s3** elements and different

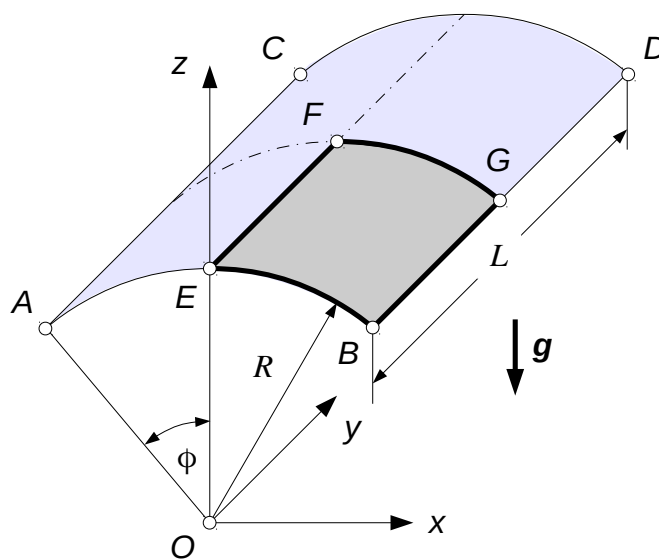


Figure 2.7-1: Scordelis-Lo Roof

mesh sizes to study the effect of the discretization on the results.

The structure is supported in x- and z-direction at the edges  $AB$  and  $CD$  and loaded by its weight. Because of symmetry only one quarter of the roof need be modeled, with symmetry boundary conditions applied at the edges  $EF$  and  $FG$ .

This example, known as the Scordelis-Lo roof<sup>1</sup>, is a standard benchmark for shell finite elements. The data and the reference solution (vertical displacement  $w_G$  at point  $G$ ) are taken from MacNeal<sup>2</sup>, but converted to SI units.

Data:  $L/2 = R = 7620$  mm, thickness  $t = 76.2$  mm,  $\phi = 40^\circ$ , Young's modulus  $E = 2.0684 \cdot 10^4$  MPa, Poisson's ratio  $\nu = 0$ , weight per area  $p = 4.309 \cdot 10^{-3}$  MPa, gravity acceleration  $g = 9.81$  m/s<sup>2</sup>, reference solution  $w_{Gref} = -92.17$  mm

### Model Definition

As the geometry of this problem is very simple, the model can be defined in GNU Octave. Because we want to use the model in different analyses, we define the model in a function that has as input arguments the number of elements along the y-axis and the element type. The number of elements along the circumference is identical to the number of elements along the y-axis. The output argument is a structure with the model definition.

The function is defined in file `model.m`:

```
function mdl = model(ne, etype)

# Input   ne           Number of elements along one direction
#         etype        Element name
# Output  mdl          Structure with model definition
#
# The function generates the model of the Scordelis-Lo roof.
#
# Node and element numbers increase by one in circumferential
# direction and by row increments in longitudinal direction.
#
# -----
# Data (N, mm)

R      =      7620;    % Radius
L      =      7620;    % Length of model
t      =      76.2;    % Thickness
phiE   =      40;     % Angle in degrees

E      = 2.0684e4;     % Young's modulus
ny     =      0;      % Poisson's ratio
```

1 Scordelis, A.C. and K.S. Lo, Computer analysis of cylindrical shells, *J. Am. Concr. Inst.* 61, 1964, pp. 539-561

2 MacNeal, R.H., *Finite Elements: Their Design and Performance*, Marcel Dekker, 1994

```

pz    = 4.309e-3; % Weight per area
g     = 9810;    % Gravity acceleration

# Nodal point data

np    = ne + 1;   % Nodes in one row
incp  = np;      % Increment of nodal point identifiers

# Model type and subtype

mdl = struct("type", "solid", "subtype", "3d");

# Geometry and material

geom = struct("t", t);

```

The mass density is computed from the weight per area:

$$\rho = \frac{p}{t g}$$

Thus, the weight load can subsequently be defined by specifying the gravity acceleration. Alternatively, we could divide the weight per area by the thickness only and define a unit acceleration.

```

mat = struct("type", "iso", "E", E, "ny", ny,
            "rho", pz / (t * g));

# Nodes

phi = linspace(0, phiE * pi / 180, np);
x   = R * sin(phi);
y   = linspace(0, L, np);
z   = R * cos(phi);

k = 1;

for n = 1 : np
    for m = 1 : np
        nodes(k) = struct("id", k++,
                        "coor", [x(m), y(n), z(m)]);
    end
end

mdl.nodes = nodes;

```

Because we want to print the displacements of the nodal points along edge *FG* and access the displacements at node *G* for comparison with the reference solution. To this end, we define two node sets **FG** and **G**.

```

nofnod = np * np;
mdl.nset.FG = (nofnod - np + 1) : nofnod; % Nodes along FG

```

```

mdl.nset.G = mdl.nset.FG(end);          % Point G

# Elements

k = 1;

switch etype
case "s4"

    enode1 = [1, 2, 2 + incp, 1 + incp];

    for n = 1 : ne
        enodes = enode1;
        for m = 1 : ne
            elem(k) = struct("id", k++, "type", etype,
                             "nodes", enodes++,
                             "geom", geom, "mat", mat);
        endfor
        enode1 += incp;
    endfor

```

As we want to plot the stress resultants of the elements along edge *FG*, we define an element set **FG** containing these elements. If **s4** elements are used, the elements along the edge *FG* are just the last **ne** elements, see Figure 2.7-2.

```

mdl.eset.FG = (elem(end).id - ne + 1) : elem(end).id;

case "s3"

    enode1 = [1, 2, 1 + incp]; enode2 = [2 + incp, 1 + incp, 2];

    for n = 1 : ne
        enodea = enode1; enodeb = enode2;
        for m = 1 : ne
            elem(k) = struct("id", k++, "type", etype,
                             "nodes", enodea++,
                             "geom", geom, "mat", mat);
            elem(k) = struct("id", k++, "type", etype,
                             "nodes", enodeb++,
                             "geom", geom, "mat", mat);
        endfor
        enode1 += incp; enode2 += incp;
    endfor

```

If **s3** elements are used, each of the quadrangles along the edge *FG* is subdivided into two triangles, see Figure 2.7-3. As the stress points of these two triangles have different y-coordinates, only the first is used for output to avoid oscillations.

```

FG = (elem(end).id - 2 * ne + 1) : 2 : elem(end).id;

```



```

    mdl.eset.FG = FG;

endswitch

mdl.elements = elem;

# Constraints

k = 1;

% at y = 0: u = w = 0

for m = 1 : np
    fix(k++) = struct("id", m, "dofs", [1, 3]);
endfor

% at phi = 0: symmetry

n2 = 1 + ne * incp;
for n = 1 : incp : n2
    fix(k++) = struct("id", n, "dofs", [1, 5, 6]);
endfor

% at y = L: symmetry

m1 = n2; m2 = m1 + ne;
for m = m1 : m2
    fix(k++) = struct("id", m, "dofs", [2, 4, 6]);
endfor

mdl.constraints.prescribed = fix;

# Load

```

The weight equals the inertia loads due to an acceleration of 1 g in positive z-direction.

```
mdl.loads.inertia = struct("data", [0, 0, g]);
```

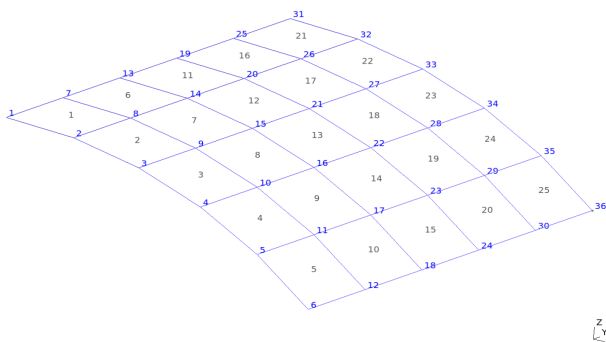


Figure 2.7-2: Roof, **s4** Mesh

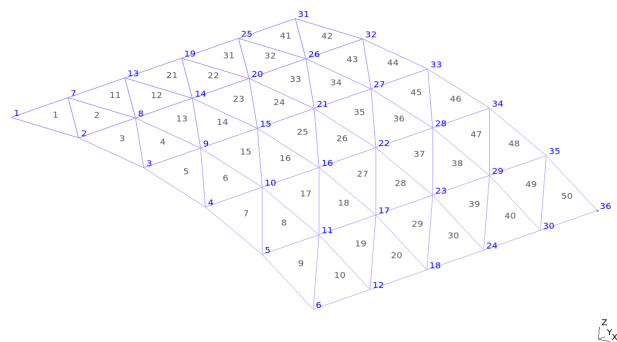


Figure 2.7-3: Roof, **s3** Mesh

**endfunction**

Typical meshes generated by this function can be seen in Figures 2.7-2 and 2.7-3.

Analysis

GNU Octave script `roof.m` performs a detailed analysis of the roof, including computation of displacements, stresses and stress resultants.

First, the discretization is defined interactively. Next, function `model` is called to create the model.

```
# Example: Scordelis-Lo roof
#
# Detailed analysis of one discretization
#
# -----

elts = {"s3", "s4"};    % List of admissible element types

# Reference solution

wref = -92.17;

# Discretization data

etype = input("Enter element type (s3 or s4): ", "s");
ne     = input("Enter number of elements in one direction: ");

if (~ ismember(etype, elts))
    error("Element type %s not supported\n", etype);
end
if (ne <= 0)
    error("Number of elements must be strictly positive\n");
end

# Output file

fid = fopen([etype, ".res"], "wt");

fprintf(fid, "Discretization: ");
fprintf(fid, "%d %s elements in one direction\n\n",
        ne, etype);

# Model

mdl = model(ne, etype);
```

Subsequently, the component is created and the mesh together with the axes is exported. Exporting of the axes allows to visualize the local coordinate systems of the elements, see Figure 2.7-4. Interpretation of the results is more

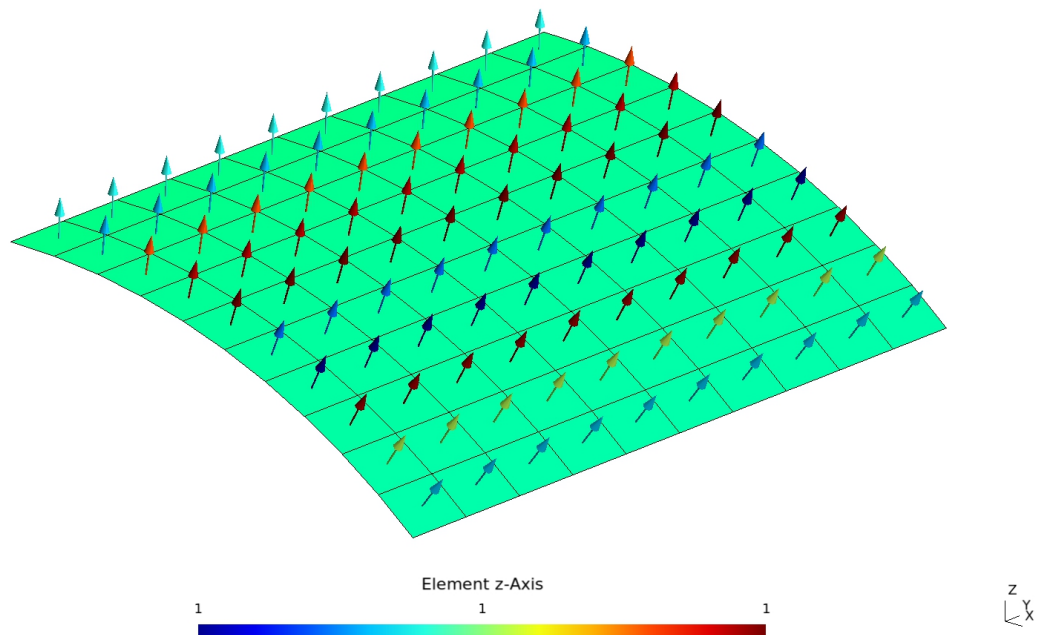


Figure 2.7-4: Roof, Shell Normal Vectors

convenient if the normal vectors are consistent.

#### # Create and export component

```
roof = mfs_new(fid, mdl);
mfs_export([etype, ".msh"], "msh", roof,
           "mesh", "mesh", "axes");
```

Next, the stiffness and mass matrix and the displacements are computed. Please note that the mass matrix is needed to compute the inertia loads.

#### # Analysis

```
roof = mfs_stiff(roof);
roof = mfs_mass(roof, "consistent");
roof = mfs_statresp(roof);
```

Function **mfs\_print** is used to write the displacements of the nodal points along edge *FG* to the output file. Please note that the item **"disp"** is the first element of cell array **{"disp", "FG"}**. The second element, **"FG"**, is the name of the set containing the nodal points along edge *FG*.

For comparison with the reference solution, function **mfs\_getresp** is used to retrieve the displacement vector at point *G*. The last input argument, **"G"**, of this function is the name of the node set containing the nodal point *G*.

The z-component of the displacement vector is divided by the reference solu-

tion and written to the output file. In addition, the complete deformation is exported for postprocessing.

```
# Displacements of free edge FG

mfs_print(fid, roof, "statresp", {"disp", "FG"});

dsp = mfs_getresp(roof, "statresp", "disp", "G");
wrel = dsp(3) / wref;
fprintf(fid, "\nNormalized displacement: uz = %7.4f\n", wrel);

# Export displacements

mfs_export([etype, ".dsp"], "msh", roof, "statresp", "disp");
```

Next, the element results, i.e. the stresses and stress resultants, are computed, printed to the output file and exported for postprocessing. Printed output is restricted to the elements in set **FG**.

```
# Element results

roof = mfs_results(roof, "statresp", "element");
mfs_print(fid, roof, "statresp", {"stress", "FG"},
          {"resultant", "FG"});
mfs_export([etype, ".sig"], "msh", roof, "statresp", "stress");
```

Finally, function **mfs\_getresp** is used to retrieve the stress resultants of the elements along edge *FG*. The last input argument, "**FG**", is the name of the set containing these elements.

The output argument, **rslt**, is a cell array of structures, each structure containing the results of one element. The fields of these structures are described in Section 4.2.2 of the User Manual.

Because all elements are of the same type, all structures contain the same fields. Hence, the GNU Octave function **cell2mat** can be used to convert the cell array of structures **rslt** to the structure array **r**. Arrays can be obtained from the different fields of the structure array by using brackets (**[...]**). For example, **[r.coor]** is a one-dimensional array containing the coordinates of all stress points. GNU Octave function **reshape** is used to convert this array into the two-dimensional matrix of coordinates, **coor(3, :)**. Subsequently, the angles **phi(:)** corresponding to the stress points are computed from the x- and z-coordinates and converted from radians to degrees.

```
# Stress resultants along edge y = L

rslt = mfs_getresp(roof, "statresp", "resultant", "FG");
r = cell2mat(rslt);
coor = reshape([r.coor], 3, length(eset));
phi = 180 * atan2(coor(1, :), coor(3, :)) / pi;
```

```

titletxt = sprintf("%s elements, %d intervals",
                   etype, ne);

```

First, the normal forces per length in  $x_E$ - and  $y_E$ -direction of the element coordinate system are plotted as a function of the angle. Again, brackets are used to get arrays from fields of a structure array.

```

figure(1, "position", [100, 100, 750, 500],
       "paperposition", [0, 0, 15, 10]);
plot(phi, [r.Nx], "color", "green",
      phi, [r.Ny], "color", "red");
title(titletxt);
legend("N_x", "N_y", "location", "northwest");
grid;
xlabel('\phi [°]'); ylabel('[N/mm]');

print("N.svg", "-dsvg");

```

Next, the bending moments  $M_x$  and  $M_y$  per length are plotted. Please note that these are the moments per length resulting from the stresses  $\sigma_x$  and  $\sigma_y$  respectively, i.e.  $M_x$  is a moment per length about the  $y_E$ -axis of the element coordinate system and  $M_y$  is a moment per length about the negative  $x_E$ -axis of the element coordinate system, see Section 4.2.2 of the User Manual.

```

figure(2, "position", [200, 100, 750, 500],
       "paperposition", [0, 0, 15, 10]);
plot(phi, [r.Mx] / 1000, "color", "green",
      phi, [r.My] / 1000, "color", "red");
title(titletxt);
legend("M_x", "M_y", "location", "northeast");
grid;
xlabel('\phi [°]'); ylabel('[Nm/mm]');

print("M.svg", "-dsvg");

fclose(fid);

```

## Results

The output file contains the displacements along edge  $FG$ , the normalized displacement  $w_G/w_{Gref}$  as well as the stresses and the stress resultants along edge  $FG$ .

Discretization: 10 s4 elements in one direction

Mefisto 2.7: Building new component from input "mdl"

Model Type = solid, Model Subtype = 3d

Number of nodes = 121, Number of elements = 100

Number of element types = 1

Number of global degrees of freedom = 726

```

Number of local      degrees of freedom = 640
Number of prescribed degrees of freedom = 86
Number of dependent degrees of freedom = 0

Number of load cases = 1

```

-----

Component "roof"

Displacements of loadcase 1

node	ux	uy	uz	rx	ry	rz
111	0.000e+00	0.000e+00	1.392e+01	0.000e+00	0.000e+00	0.000e+00
112	-7.602e-02	0.000e+00	1.223e+01	0.000e+00	6.394e-03	0.000e+00
113	-6.179e-01	0.000e+00	7.227e+00	0.000e+00	1.251e-02	0.000e+00
114	-2.046e+00	0.000e+00	-7.819e-01	0.000e+00	1.808e-02	0.000e+00
115	-4.692e+00	0.000e+00	-1.133e+01	0.000e+00	2.283e-02	0.000e+00
116	-8.764e+00	0.000e+00	-2.382e+01	0.000e+00	2.656e-02	0.000e+00
117	-1.432e+01	0.000e+00	-3.756e+01	0.000e+00	2.914e-02	0.000e+00
118	-2.129e+01	0.000e+00	-5.182e+01	0.000e+00	3.057e-02	0.000e+00
119	-2.949e+01	0.000e+00	-6.601e+01	0.000e+00	3.103e-02	0.000e+00
120	-3.870e+01	0.000e+00	-7.967e+01	0.000e+00	3.092e-02	0.000e+00
121	-4.880e+01	0.000e+00	-9.259e+01	0.000e+00	3.077e-02	0.000e+00

Normalized displacement: uz = 1.0046

-----

It can be seen that the vertical displacement of point G, obtained with a mesh consisting of 10 by 10 **s4** elements, is in very good agreement with the reference solution, the difference being only 0.46 %. Figure 2.7-5 shows the deformed shape of the roof.

Stresses are output at the stress points of the lower and the upper side of the

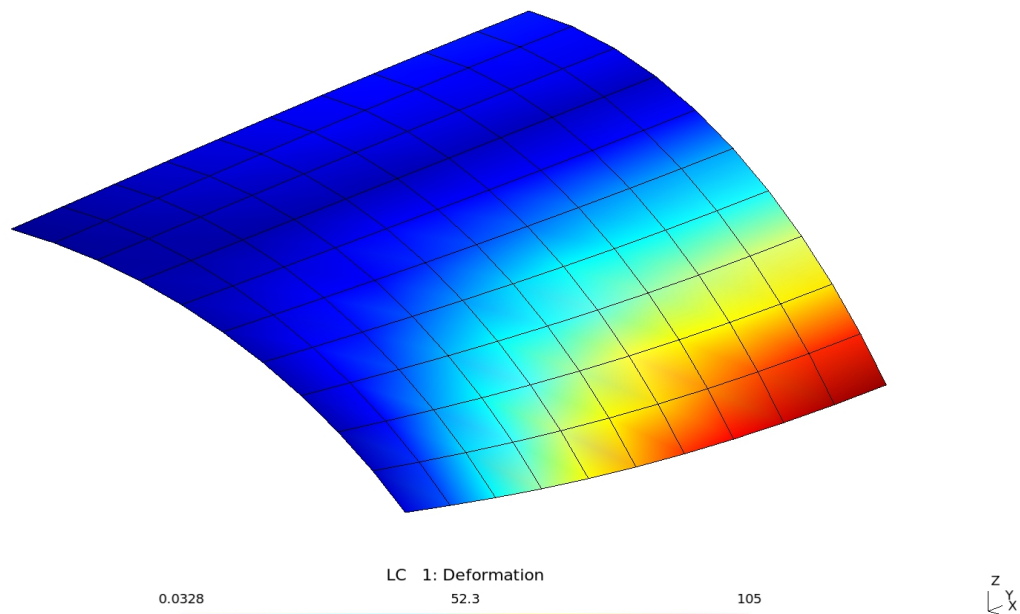


Figure 2.7-5: Roof, Deformed Structure

element, the normal vector pointing from the lower to the upper side. The **s3** and the **s4** elements have only one stress point which is the midpoint of the element. The stresses refer to the element coordinate system. In this example, the  $x_E$ -axis of the **s4** elements is tangential to the circumference, and the  $y_E$ -axis is parallel to the  $y$ -axis. The  $z_E$ -axis is always parallel to the element normal vector.

Component "roof"

Stresses of loadcase 1

Shell elements:

element	point	side	sigx	sigy	tauxy
<hr/>					
91	1	upper	8.767e+00	7.619e-02	4.476e-02
		lower	-1.006e+01	-7.601e-01	-3.747e-02
92	1	upper	8.371e+00	-4.731e-01	1.324e-01
		lower	-9.644e+00	-1.077e+00	-1.077e-01
93	1	upper	7.586e+00	-1.456e+00	2.143e-01
		lower	-8.812e+00	-1.615e+00	-1.650e-01
94	1	upper	6.440e+00	-2.647e+00	2.868e-01
		lower	-7.581e+00	-2.182e+00	-2.042e-01
95	1	upper	5.000e+00	-3.729e+00	3.465e-01
		lower	-6.009e+00	-2.508e+00	-2.247e-01
96	1	upper	3.393e+00	-4.313e+00	3.904e-01
		lower	-4.218e+00	-2.249e+00	-2.306e-01
97	1	upper	1.813e+00	-3.957e+00	4.163e-01
		lower	-2.411e+00	-1.007e+00	-2.299e-01
98	1	upper	5.098e-01	-2.187e+00	4.218e-01
		lower	-8.674e-01	1.659e+00	-2.334e-01
99	1	upper	-2.441e-01	1.488e+00	4.042e-01
		lower	9.848e-02	6.218e+00	-2.520e-01
100	1	upper	-2.249e-01	7.564e+00	3.598e-01
		lower	1.985e-01	1.316e+01	-2.955e-01

Also the stress resultants are output at the stress points. They refer to the element coordinate system. The first three columns contain the membrane forces and bending moments per unit length, the fourth column the transverse shear forces per length.

Stress resultants of loadcase 1

Shell elements:

element	point	Nx / Mx	Ny / My	Nxy / Mxy	Qx / Qy
<hr/>					
91	1	-4.928e+01	-2.606e+01	2.774e-01	-4.745e-01
		9.110e+03	4.047e+02	3.979e+01	3.578e-03
92	1	-4.852e+01	-5.907e+01	9.386e-01	-1.414e+00
		8.717e+03	2.924e+02	1.162e+02	1.921e-03
93	1	-4.673e+01	-1.170e+02	1.876e+00	-2.316e+00
		7.935e+03	7.694e+01	1.835e+02	-9.313e-04
94	1	-4.350e+01	-1.840e+02	3.144e+00	-3.116e+00
		6.784e+03	-2.247e+02	2.376e+02	-4.250e-03
95	1	-3.843e+01	-2.376e+02	4.638e+00	-3.716e+00
		5.327e+03	-5.910e+02	2.764e+02	-7.347e-03
96	1	-3.142e+01	-2.500e+02	6.091e+00	-3.984e+00
		3.683e+03	-9.989e+02	3.005e+02	-9.710e-03
97	1	-2.282e+01	-1.891e+02	7.100e+00	-3.789e+00
		2.044e+03	-1.428e+03	3.127e+02	-1.134e-02
98	1	-1.363e+01	-2.010e+01	7.177e+00	-3.044e+00
		6.664e+02	-1.861e+03	3.170e+02	-1.224e-02
99	1	-5.548e+00	2.936e+02	5.797e+00	-1.751e+00
		-1.658e+02	-2.289e+03	3.175e+02	-1.326e-02

100	1	-1.008e+00	7.894e+02	2.450e+00	-6.165e-02
		-2.049e+02	-2.705e+03	3.171e+02	-1.189e-02

Figures 2.7-6 and 2.7-7 show the stress resultants along the edge  $FG$ . They are in good agreement with the results reported by Scordelis and Lo. At the free edge ( $\phi = 40^\circ$ ) both  $N_x$  and  $M_x$  are zero.

Figure 2.7-8 compares the performance of the **s4** and **s3** elements. It shows the normalized displacement at point G as a function of the number of intervals. The data have been computed with GNU Octave script `roof_loop.m` and plotted with GNU Octave script `plot_loop.m`.

It can be seen that in this problem the displacements obtained with the **s4** element converge from above and those obtained with the **s3** element from below. The performance of the **s4** element is much better than that of the **s3** element. Already with 10 intervals, the solution obtained with **s4** elements differs by less than 1 % from the reference solution, whereas about 50 intervals are needed to obtain the same accuracy with **s3** elements. In addition, the **s3** element tends to lock if the mesh is very coarse. Thus, **s4** elements should be preferred. **s3** elements should be used only if absolutely necessary.

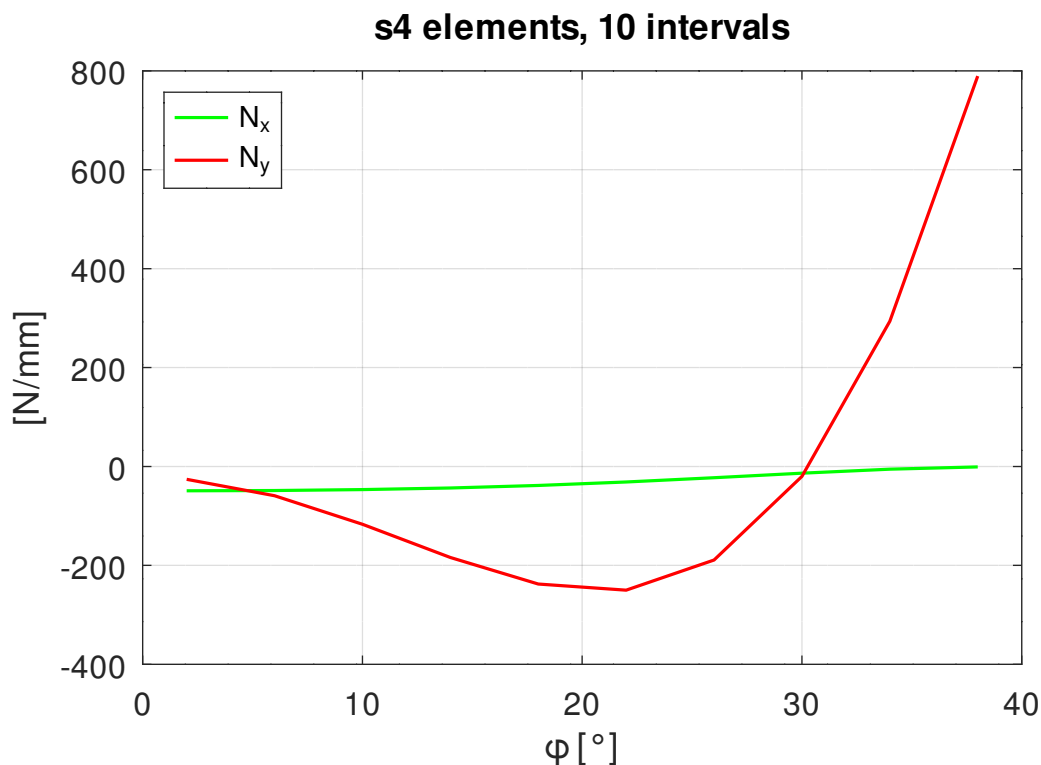


Figure 2.7-6: Roof, Normal Forces per Length



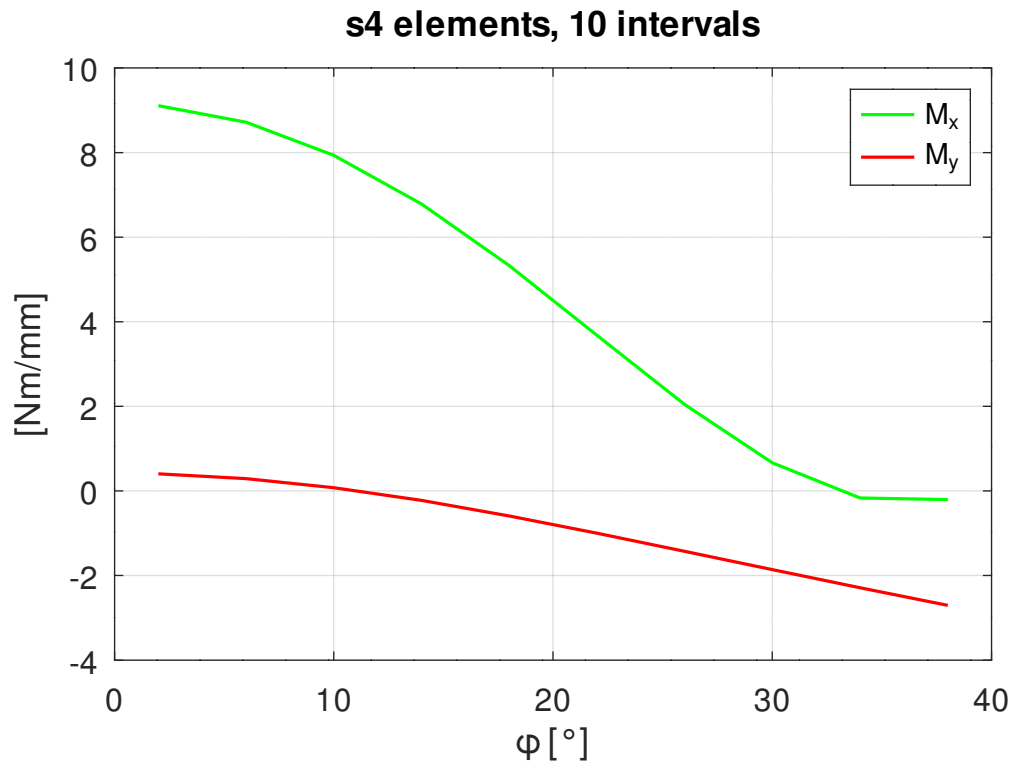


Figure 2.7-7: Roof, Bending Moments per Length

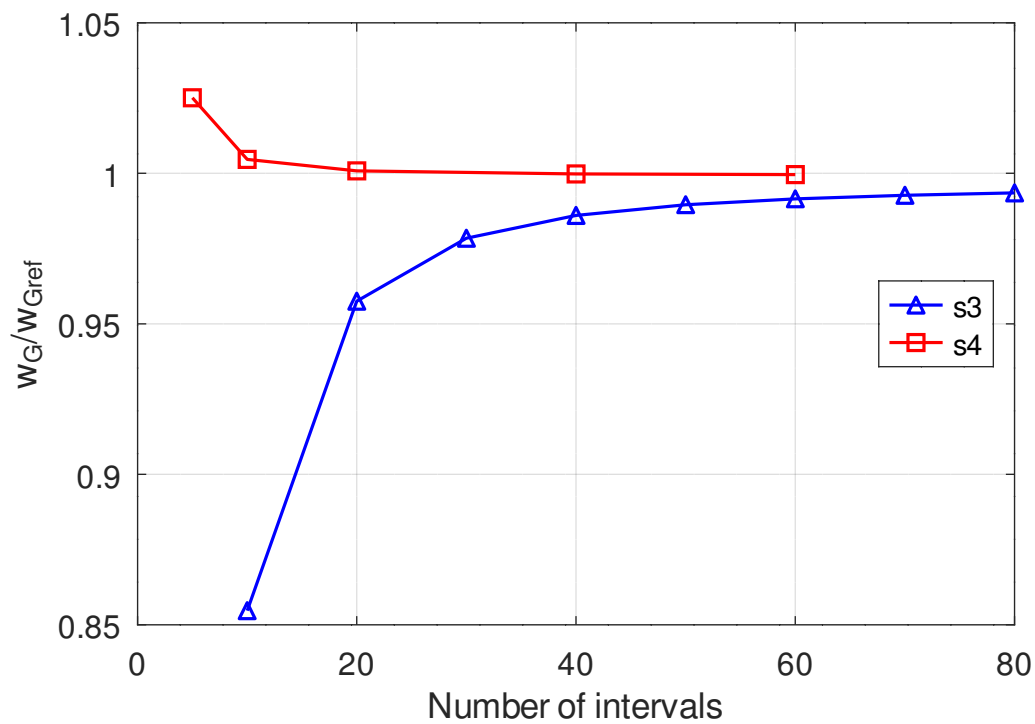


Figure 2.7-8: Roof, Convergence Study

### 3 Normal Modes Analysis

#### 3.1 2-dimensional Frame

##### Summary

Directory:	exa/solid/freevib/frame2d
Objectives:	<ul style="list-style-type: none"> <li>• learn how to define 2-dimensional beam elements</li> <li>• learn how to run a normal modes analysis</li> <li>• learn how to compute and interpret modal effective masses</li> <li>• learn how to do pre- and postprocessing using Mefisto functions</li> </ul>
Dimension:	2
Elements:	<b>b2</b>
Loads:	none
Functions:	<b>mfs_line, mfs_new, mfs_stiff, mfs_mass, mfs_freevib, mfs_print, mfs_plot, mfs_meffmass</b>

##### Problem Description

Compute the first 25 normal modes and their modal effective masses of the frame structure shown in Figure 3.1-1.

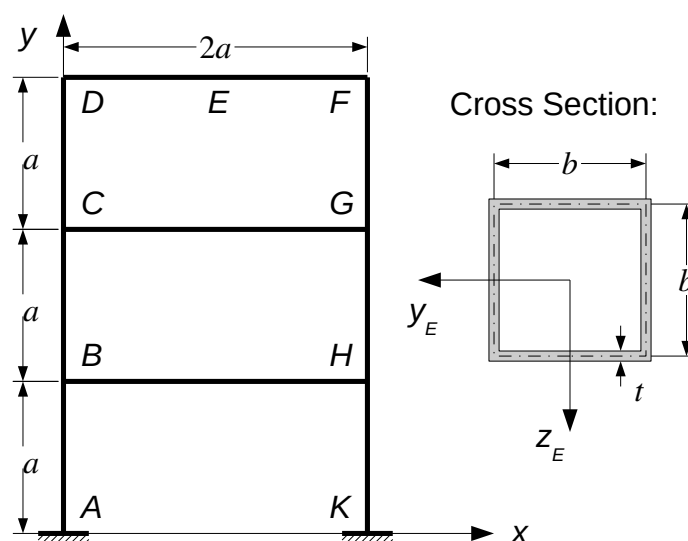


Figure 3.1-1: Frame Structure

Data:  $a = 2500$  mm,  $b = 0.01a$ ,  $t = 0.1b$ , Young's modulus  $E = 210$  GPa, mass density  $\rho = 7850$  kg/m<sup>3</sup>

### Model Definition

In a dynamic analysis, beam elements do not provide exact results. The element size strongly influences the results. It should be small enough so as to give a good resolution of the smallest wavelength of interest. Hence, the element size depends on the highest frequency of interest. If no analytical results are available to estimate the wave length, whether the element length is adequate can be checked by looking at the mode shapes.

In this example, we define the model with a variable number of elements directly in GNU Octave, without using Gmsh. File `frame.m` starts with the definition of the model data:

```
fid = fopen("frame.res", "wt");

# Data (N, mm)

nmodes = 25; % Number of modes to be computed

a      = 2500; % Length
nela   = 20; % Number of elements per length

E      = 210000; % Young's modulus
rho    = 7.85E-9; % Mass density
```

The units used are N and mm. Thus, the unit of the mass is not an independent unit but has to be expressed in terms of N, mm and s:

$$1 \text{ N} = 1 \frac{\text{kg m}}{\text{s}^2} \rightarrow 1 \text{ kg} = 1 \frac{\text{N s}^2}{\text{m}} = 10^{-3} \frac{\text{N s}^2}{\text{mm}} \rightarrow 1 \frac{\text{N s}^2}{\text{mm}} = 10^3 \text{ kg} = 1 \text{ t}$$

This computation shows that the mass unit to be used is t. Then, the unit of the mass density is t/mm<sup>3</sup>.

Next, we compute the cross section properties. For a 2-dimensional beam element, only the cross section area  $A$  and the moment of inertia  $I_z$  are needed. In case of a thin-walled square, these properties can be computed from

$$A = 4bt, \quad I_z = 2 \left( \frac{b^3 t}{12} + \frac{b^2}{4} bt \right) = \frac{2}{3} b^3 t.$$

The following code computes the cross section properties and defines the model type and subtype as well as the geometrical and the material data:

```
# Beam cross section data: Thin-walled square section
```

```

b =      0.01 * a; % Length of edge of square
t =      0.1 * b; % Thickness
A =      4 * b * t; % Area
I = 2 * b^3 * t / 3; % Moment of inertia

# Model definition
# -----

model = struct("type", "solid", "subtype", "2d");

# Geometry and material

geom = struct("A", A, "I", I);
mat = struct("type", "iso", "E", E, "rho", rho);

```

Next, we define the identifiers of the nodal points corresponding to points A to K and the coordinates of the nodal points corresponding to points A, D, F and K. To define the remaining nodal points as well as the elements we can use function `mfs_line`. The nodal points created are added to structure array `nodes`. Elements, however, are not added to an existing structure. Thus, each call to function `mfs_line` generates a separate structure array, and the complete definition of the elements is obtained by concatenating these structure arrays.

```

# Outer frame

idA = 1;
idB = idA + nela;
idC = idB + nela;
idD = idC + nela;
idE = idD + nela;
idF = idE + nela;
idG = idF + nela;
idH = idG + nela;
idK = idH + nela;

nodes(1).id = idA; nodes(1).coor = [0, 0];
nodes(2).id = idD; nodes(2).coor = [0, 3 * a];
nodes(3).id = idF; nodes(3).coor = [2 * a, 3 * a];
nodes(4).id = idK; nodes(4).coor = [2 * a, 0];

[nodes, elem1] = mfs_line(nodes, idA, idD,
                        (idA + 1) : (idD - 1),
                        1 : (3 * nela), "b2", geom, mat);
[nodes, elem2] = mfs_line(nodes, idD, idF,
                        (idD + 1) : (idF - 1),
                        (3 * nela + 1) : (5 * nela),
                        "b2", geom, mat);
[nodes, elem3] = mfs_line(nodes, idF, idK,
                        (idF + 1) : (idK - 1),
                        (5 * nela + 1) : (8 * nela),
                        "b2", geom, mat);

```

```
# Cross beams BH and CG

id1 = idK + 1;
id2 = idK + 2 * nela - 1;
[nodes, elem4] = mfs_line(nodes, idB, idH, id1 : id2,
                        (8 * nela + 1) : (10 * nela),
                        "b2", geom, mat);

id1 = id2 + 1;
id2 = id1 + 2 * nela - 2;
[nodes, elem5] = mfs_line(nodes, idC, idG, id1 : id2,
                        (10 * nela + 1) : (12 * nela),
                        "b2", geom, mat);

model.nodes      = nodes;
model.elements   = [elem1, elem2, elem3, elem4, elem5];
```

Finally, we define the constraints. The structure is clamped at points A and K. The use of a cell array for the nodal point identifiers causes GNU Octave to create a structure array.

```
# Constraints

cst = struct("id", {idA, idK}, "dofs", [1, 2, 3]);
model.constraints.prescribed = cst;
```

In a normal modes analysis, no loads need be defined.

### Analysis

First, we use function **mfs\_new** to create a new component which we call

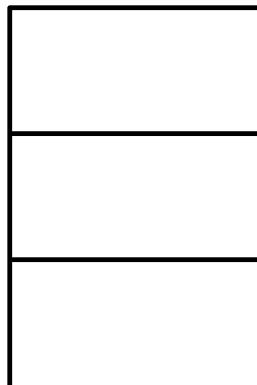


Figure 3.1-2: Frame, Finite Element Model

**frame.** Next, we use function `mfs_plot` to plot the component so that we can see if the model is defined correctly. Figure 3.1-2 shows the resulting plot.

Subsequently, we compute the stiffness and the mass matrix using functions `mfs_stiff` and `mfs_mass`. As we do not define the mass type, a lumped mass matrix is computed. Then, function `mfs_freevib` computes the normal modes. In most cases, the default settings can be used so that only the number of modes to be computed needs to be specified.

```
# Analysis
# -----

# Create and plot component

frame = mfs_new(fid, model);
mfs_plot(frame, "paperposition", [0, 0, 10, 10]);
print("frame.svg", "-dsvg");

# Analysis

frame = mfs_stiff(frame);
frame = mfs_mass(frame);

frame = mfs_freevib(frame, nmodes);
mfs_print(fid, frame, "modes", "freq", "reac");
```

Function `mfs_print` writes the circular frequencies, the frequencies and the reaction loads to the output file:

Mefisto 2.7: Building new component from input "model"

```
Model Type = solid, Model Subtype = 2d

Number of nodes      = 239, Number of elements = 240
Number of element types = 1
Number of global      degrees of freedom = 717
Number of local      degrees of freedom = 711
Number of prescribed degrees of freedom = 6
Number of dependent  degrees of freedom = 0
```

-----

Component "frame"

Natural frequencies:

Mode	Circ. Frequency	Frequency
1	5.60380	0.89187 Hz
2	19.35801	3.08092 Hz
3	32.67707	5.20072 Hz
4	37.32940	5.94116 Hz
5	37.77559	6.01217 Hz
6	40.43086	6.43477 Hz
7	88.61798	14.10399 Hz
8	103.86408	16.53048 Hz
9	112.26347	17.86729 Hz
10	118.74574	18.89897 Hz
11	144.63595	23.01953 Hz

12	153.30774	24.39968 Hz
13	177.67896	28.27849 Hz
14	179.52008	28.57151 Hz
15	214.84738	34.19402 Hz
16	221.66219	35.27863 Hz
17	237.26044	37.76117 Hz
18	243.35692	38.73146 Hz
19	341.41507	54.33790 Hz
20	372.14269	59.22835 Hz
21	389.57034	62.00204 Hz
22	400.22764	63.69821 Hz
23	445.73988	70.94171 Hz
24	458.34497	72.94787 Hz
25	494.94794	78.77341 Hz

Reaction loads of mode 1

node	Fx	Fy	Mz
1	-3.351e+00	-4.984e+00	6.020e+03
161	-3.351e+00	4.984e+00	6.020e+03
Res.	-6.701e+00	1.461e-11	3.696e+04

Reaction loads of mode 2

node	Fx	Fy	Mz
1	1.565e+01	-7.322e+00	-2.135e+04
161	1.565e+01	7.322e+00	-2.135e+04
Res.	3.130e+01	6.392e-11	-6.093e+03

Reaction loads of mode 3

node	Fx	Fy	Mz
1	-6.268e+00	-2.789e+01	5.103e+03
161	6.268e+00	-2.789e+01	-5.103e+03
Res.	1.217e-10	-5.579e+01	-1.395e+05

...                      ...                      ...

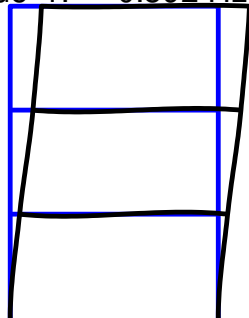
The following commands generate a plot which shows six selected normal modes. To get several plots into the same figure, we first create the figure and supply the figure handle to function `mfs_plot`. Figure 3.1-3 shows the resulting picture.

# Plot some modes

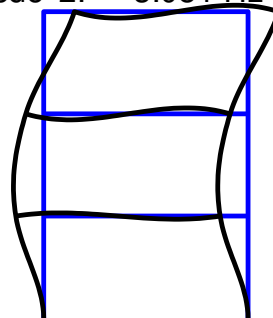
```
h = figure("position", [100, 100, 750, 1000],
           "paperposition", [0, 0, 16, 20]);
subplot(3, 2, 1);
    mfs_plot(frame, "deform", 2, "modes", 1, "figure", h);
subplot(3, 2, 2);
    mfs_plot(frame, "deform", 2, "modes", 2, "figure", h);
subplot(3, 2, 3);
    mfs_plot(frame, "deform", 2, "modes", 3, "figure", h);
subplot(3, 2, 4);
    mfs_plot(frame, "deform", 2, "modes", 4, "figure", h);
subplot(3, 2, 5);
    mfs_plot(frame, "deform", 2, "modes", 6, "figure", h);
subplot(3, 2, 6);
    mfs_plot(frame, "deform", 2, "modes", 16, "figure", h);
```

```
print("modes.svg", "-dsvg", "-F:12");  
# Modal effective masses  
mfs_meffmass(fid, frame);  
fclose(fid);
```

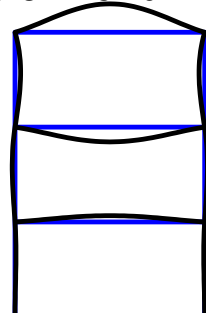
Mode 1: 0.892 Hz



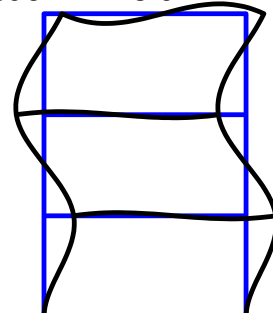
Mode 2: 3.081 Hz



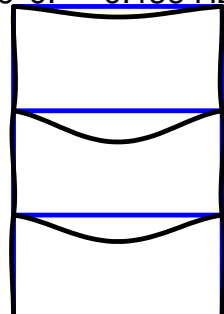
Mode 3: 5.201 Hz



Mode 4: 5.941 Hz



Mode 6: 6.435 Hz



Mode 16: 35.279 Hz

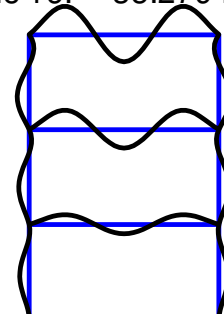


Figure 3.1-3: Frame, Some Normal Modes



Finally, function `mfs_meffmass` computes the modal effective masses and writes them to the output file:

-----  
Modal effective masses of component "frame"

Coordinates of reference point: 0.0000, 0.0000

mode	frequency	tx	ty	rz	sum tx	sum ty	sum rz
1	0.89 Hz	7.7673e-01	1.1129e-26	6.8557e-01	0.77673	0.00000	0.68557
2	3.08 Hz	1.1901e-01	4.7388e-28	1.3083e-04	0.89575	0.00000	0.68570
3	5.20 Hz	3.6541e-25	4.6557e-02	8.4419e-03	0.89575	0.04656	0.69414
4	5.94 Hz	4.0127e-02	3.9940e-25	1.6052e-03	0.93587	0.04656	0.69575
5	6.01 Hz	4.3009e-24	3.8293e-03	6.9433e-04	0.93587	0.05039	0.69644
6	6.43 Hz	4.7084e-27	3.2008e-01	5.8037e-02	0.93587	0.37047	0.75448
7	14.10 Hz	4.2024e-03	3.5472e-26	2.1654e-03	0.94008	0.37047	0.75665
8	16.53 Hz	3.9111e-03	1.2446e-27	6.7262e-05	0.94399	0.37047	0.75671
9	17.87 Hz	7.2473e-04	2.7121e-27	2.2416e-02	0.94471	0.37047	0.77913
10	18.90 Hz	1.5967e-26	3.8356e-05	6.9548e-06	0.94471	0.37050	0.77914
11	23.02 Hz	4.9155e-28	3.5397e-06	6.4182e-07	0.94471	0.37051	0.77914
12	24.40 Hz	4.9368e-03	2.7575e-27	1.2647e-04	0.94965	0.37051	0.77926
13	28.28 Hz	1.1134e-02	5.3513e-26	9.2381e-07	0.96078	0.37051	0.77926
14	28.57 Hz	1.1682e-25	3.6353e-03	6.5916e-04	0.96078	0.37414	0.77992
15	34.19 Hz	8.7334e-03	4.7199e-27	1.7638e-08	0.96952	0.37414	0.77992
16	35.28 Hz	4.2524e-28	5.0970e-02	9.2419e-03	0.96952	0.42511	0.78916
17	37.76 Hz	5.2937e-28	8.4572e-03	1.5335e-03	0.96952	0.43357	0.79070
18	38.73 Hz	2.6834e-28	4.9198e-04	8.9206e-05	0.96952	0.43406	0.79079
19	54.34 Hz	8.2736e-04	6.8634e-29	3.6680e-03	0.97034	0.43406	0.79446
20	59.23 Hz	1.1887e-03	7.4647e-29	5.0107e-05	0.97153	0.43406	0.79451
21	62.00 Hz	2.5453e-04	1.9803e-28	4.7302e-05	0.97179	0.43406	0.79455
22	63.70 Hz	2.3479e-28	4.8323e-04	8.7621e-05	0.97179	0.43454	0.79464
23	70.94 Hz	4.3178e-33	2.9466e-06	5.3428e-07	0.97179	0.43455	0.79464
24	72.95 Hz	1.3238e-03	2.9339e-31	4.3769e-03	0.97311	0.43455	0.79902
25	78.77 Hz	4.0880e-03	7.1578e-30	4.7789e-05	0.97720	0.43455	0.79907

It can be seen that normal modes with motion mainly in the lateral direction have large modal effective masses in x-direction (modes 1, 2 and 4) whereas modes with motion mainly in the vertical direction have large modal effective masses in y-direction (modes 3, 6 and 16). It can be observed that modes with large modal effective masses in x-direction also have large reaction load resultants in x-direction and modes with large modal effective masses in y-direction have large reaction load resultants in y-direction.

## 3.2 2-dimensional Arc

### Summary

Directory:	exa/solid/freevib/arc2d
Objectives:	<ul style="list-style-type: none"> <li>• learn how to compute and interpret the mass properties of a 2-dimensional structure</li> <li>• learn how to postprocess normal modes using Gmsh</li> <li>• study the effect of using different element types and different mesh sizes</li> </ul>

	<ul style="list-style-type: none"> <li>study the effect of using different types of mass matrices</li> </ul>
Dimension:	2
Elements:	<b>t3, t6, q4, q8, q9</b>
Loads:	none
Functions:	<b>mfs_import, mfs_new, mfs_stiff, mfs_mass, mfs_massproperties, mfs_freevib, mfs_print, mfs_export, mfs_meffmass</b>

### Problem Description

Compute the mass properties, the first ten normal modes and their modal effective masses of the arc shown in Figure 3.2-1. Postprocess the normal modes using Gmsh. Study the effect of using different element types and different mesh sizes and compare the results obtained with a lumped mass matrix with those obtained with a consistent mass matrix.

Data:  $h = 400$  mm,  $b = 50$  mm,  $r = 100$  mm, thickness  $t = 2$  mm, Young's modulus  $E = 210$  GPa, Poisson's ratio  $\nu = 0.3$ , mass density  $\rho = 7850$  kg/m<sup>3</sup>

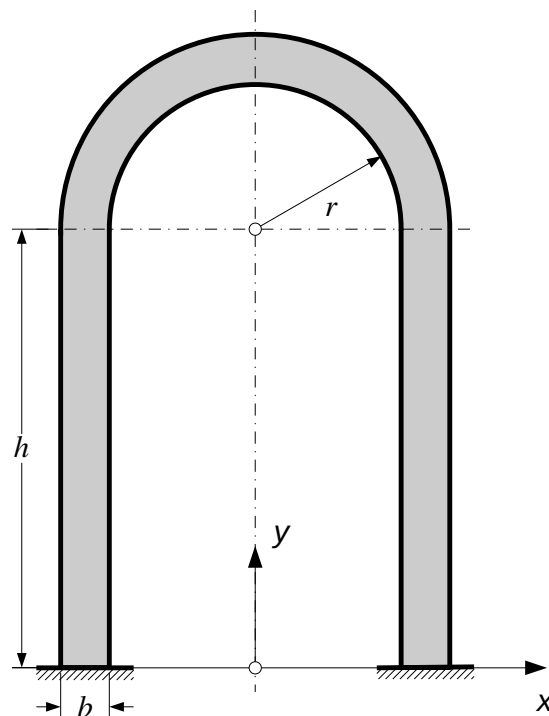


Figure 3.2-1: Arc

## Model Definition

To get a structured mesh, the geometry is subdivided into three quadrilateral surfaces, see Figure 3.2-2.

File `arc.geo` contains the commands to generate the geometry and the mesh. The commands used are described in detail in Example 2.3.

```
// Dimensions [mm]

DefineConstant [ h = 400,
                 b =  50,
                 r = 100 ];

// Meshing parameters

nb = GetValue("Number of Elements over Thickness?", 4);
nh = Floor(nb * h / b + 0.5);
na = Floor(nb * Pi * r / b + 0.5);

Mesh.ElementOrder =
GetValue("Element Order: 1 = Linear, 2 = Quadratic?", 1);
If (Mesh.ElementOrder == 2)
    Mesh.SecondOrderIncomplete =
    GetValue("Incomplete Order: 0 = no, 1 = yes?", 0);
EndIf
Mesh.RecombineAll =
GetValue("Create Quadrangles: 0 = no, 1 = yes?", 1);
Mesh.Smoothering = 1;

// Geometry

R = r + b;

Point(1) = {-R, 0, 0};
Point(2) = {-r, 0, 0};
Point(3) = {-r, h, 0};
Point(4) = {-R, h, 0};

Point(5) = {r, 0, 0};
Point(6) = {R, 0, 0};
Point(7) = {R, h, 0};
Point(8) = {r, h, 0};

Point(9) = {0, h, 0};

Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 1};

Line(5) = {5, 6};
Line(6) = {6, 7};
Line(7) = {7, 8};
```

```

Line(8) = {8, 5};

Circle( 9) = {8, 9, 3};
Circle(10) = {7, 9, 4};

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

Plane Surface(1) = {1};
Plane Surface(2) = {2};
Plane Surface(3) = {3};

// Elements

Physical Surface("Elements") = {1, 2, 3};

// Constraints

Physical Line("Constraints") = {1, 5};

// Commands to generate structured mesh

Transfinite Line{2, 4, 6, 8} = nh + 1;
Transfinite Line{1, 3, 5, 7} = nb + 1;
Transfinite Line{9, 10} = na + 1;

Transfinite Surface{1, 2, 3};

```

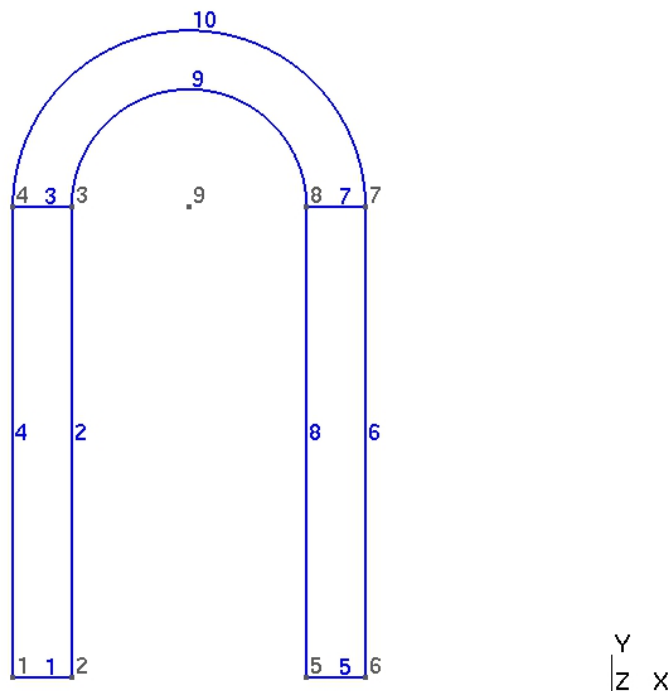


Figure 3.2-2: Arc, Geometry

The translation data can be found at the beginning of file `arc.m`. As in Examples 2.2 and 2.3, a translation table is defined that connects the Gmsh element types with the Mefisto element names. This translation table is defined in cell array `elnam`.

```
# Data (N, mm)

E    = 210000; % Young's modulus
ny   = 0.3; % Poisson's ratio
rho  = 7.85E-9; % Mass density
t    = 2;

nmodes = 10;
elnam  = {"2 = t3", "3 = q4", "9 = t6", "10 = q9", "16 = q8"};

# Type of mass matrix

masstype = {"consistent", "lumped"};
[mtp, ok] = listdlg("Name", "Select mass type",
                   "ListString", masstype,
                   "SelectionMode", "single",
                   "ListSize", [200, 50]);
if (~ ok) exit; endif

fid = fopen("arc.res", "wt");

fprintf(fid, "Mass type = %s\n\n", masstype{mtp});

# Translation data

model = struct("type", "solid", "subtype", "2d");

geom = struct("t", t);
mat = struct("type", "iso", "E", E, "ny", ny, "rho", rho);

model.Elements = struct("type", "elements",
                       "name", {elnam},
                       "geom", geom,
                       "mat", mat);

model.Constraints = struct("type", "constraints",
                          "name", "prescribed",
                          "dofs", [1, 2]);
```

## Analysis

The commands to run the analysis are mainly the same as those used in Example 3.1. In addition, function `mfs_massproperties` is used to compute the mass, the coordinates of the centre of mass and the moments of inertia. The mass properties are returned in structure `mp`. Field `mp.cm` contains the coordinates of the centre of mass which are used as input to function

**mfs\_meffmass.** Thus, the modal effective masses are computed with respect to the centre of mass.

```
# Import the msh-file
    input = mfs_import(fid, "arc.msh", "msh", model);

# Initialize component
    arc = mfs_new(fid, input);

# Stiffness and mass matrices
    arc = mfs_stiff(arc);
    arc = mfs_mass(arc, masstype{mtp});
    mp = mfs_massproperties(fid, arc);

# Free vibrations
    arc = mfs_freevib(arc, nmodes);

# Output
    mfs_print(fid, arc, "modes", "freq");
    mfs_export("arc.pos", "msh", arc, "modes", "disp");

# Modal effective masses
    mfs_meffmass(fid, arc, mp.cm);

    fclose(fid);
```

With five **q8** elements over the width and a lumped mass matrix, the output written to file **arc.res** is as follows:

```
Mass type = lumped

Reading model from file "arc.msh", MSH file version 4.1

Physical Group          Type
-----
Elements                elements
Constraints              constraints

Mefisto 2.7: Building new component from input "input"

Model Type = solid, Model Subtype = 2d

Number of nodes      = 1898,  Number of elements =   555
Number of element types = 1
Number of global      degrees of freedom =   3796
Number of local        degrees of freedom =   3752
Number of prescribed degrees of freedom =    44
Number of dependent   degrees of freedom =     0

Mass properties of component "arc"

Coordinates of reference point:      0.0000,      0.0000

Rigid body mass matrix:
```

```

9.3627e-04    0.0000e+00   -2.7375e-01
0.0000e+00    9.3627e-04   -6.3706e-14
-2.7375e-01   -6.3707e-14    1.1765e+02

```

```
Mass = 9.3627e-04, Moment of inertia = 1.1765e+02
```

```
Coordinates of center of mass:    -0.0000,    292.3846
```

```
Moment of inertia with respect to center of mass: 3.7614e+01
```

The rigid body mass matrix relates rigid body accelerations to the corresponding inertia loads. The first column contains the inertia loads due to an acceleration in x-direction, the second contains the inertia loads due to an acceleration in y-direction and the third column due to an angular acceleration about the z-axis.

The first two elements on the diagonal are the mass and the third element is the mass moment of inertia about the z-axis. The remaining elements of the matrix are the static moments from which the coordinates of the centre of mass are computed.

The output file continues with the natural frequencies and the modal effective masses:

```
-----
Component "arc"
```

```
Natural frequencies:
```

Mode	Circ. Frequency	Frequency
1	1336.48316	212.70790 Hz
2	4958.30095	789.13811 Hz
3	7448.82356	1185.51709 Hz
4	11716.21615	1864.69371 Hz
5	14943.45550	2378.32481 Hz
6	15462.11872	2460.87263 Hz
7	17732.74139	2822.25345 Hz
8	23345.16214	3715.49795 Hz
9	29596.71672	4710.46376 Hz
10	35237.35546	5608.19930 Hz

```
-----
Modal effective masses of component "arc"
```

```
Coordinates of reference point:    0.0000,    292.3846
```

mode	frequency	tx	ty	rz	sum tx	sum ty	sum rz
1	212.71 Hz	7.2737e-01	5.4154e-27	1.8660e-01	0.72737	0.00000	0.18660
2	789.14 Hz	1.0693e-25	1.6548e-02	1.2639e-25	0.72737	0.01655	0.18660
3	1185.52 Hz	1.2185e-01	5.7254e-27	2.8228e-01	0.84922	0.01655	0.46888
4	1864.69 Hz	8.3272e-28	3.2016e-01	9.0490e-32	0.84922	0.33671	0.46888
5	2378.32 Hz	2.8981e-27	4.5911e-01	4.7145e-29	0.84922	0.79582	0.46888
6	2460.87 Hz	8.0520e-03	4.7440e-27	1.5215e-01	0.85727	0.79582	0.62103
7	2822.25 Hz	4.2545e-02	5.5045e-27	1.2787e-01	0.89982	0.79582	0.74889
8	3715.50 Hz	5.1828e-28	2.6389e-02	1.3530e-27	0.89982	0.82221	0.74889
9	4710.46 Hz	2.5665e-02	5.3338e-29	5.1040e-02	0.92548	0.82221	0.79993
10	5608.20 Hz	1.3436e-28	1.2163e-02	1.3959e-28	0.92548	0.83437	0.79993

The modal effective masses show that modes 1, 3, 6, 7 and 9 are lateral modes whereas modes 2, 4, 5, 8 and 10 are vertical modes. This is confirmed by Figures 3.2-3 and 3.2-4.

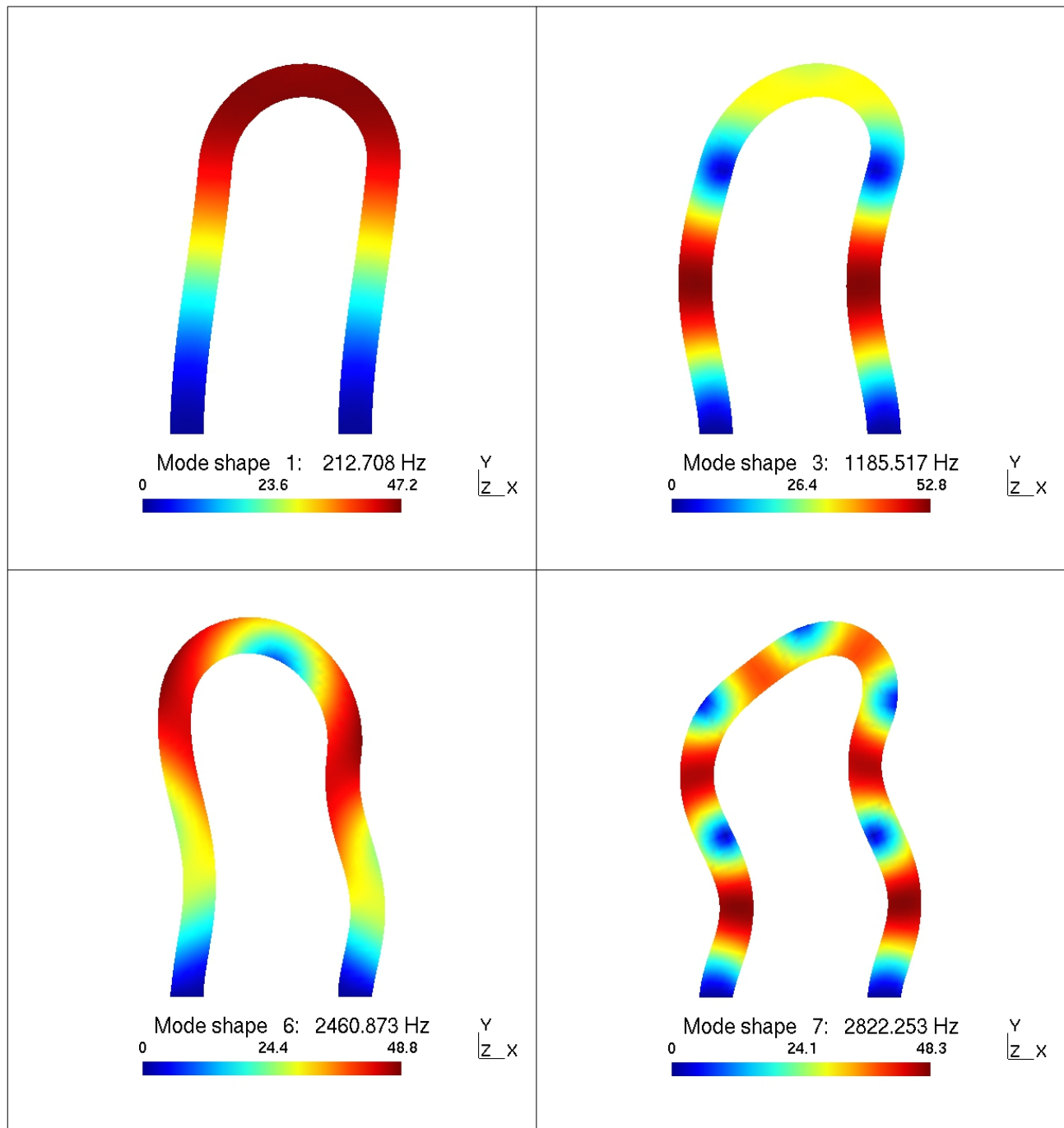


Figure 3.2-3: Arc, Some Lateral Modes



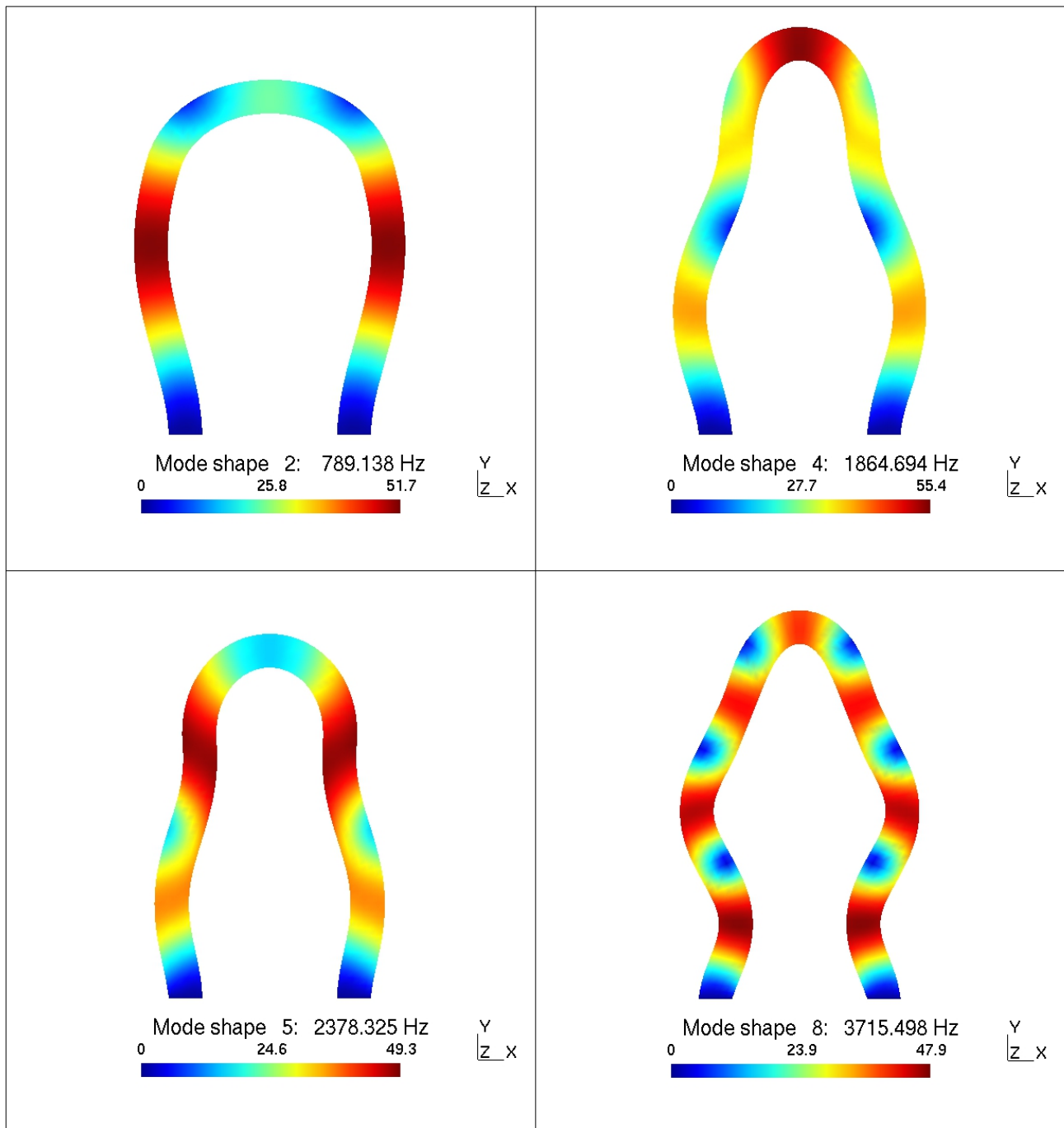


Figure 3.2-4: Arc, Some Vertical Modes

### Postprocessing

After opening file `arc.msh` and merging the results contained in file `arc.pos` all ten views are active. A single view can be selected by pressing the shift key and clicking the box in front of the view. Instead of manually adjusting the scale for each mode, you can use file `scale.gms` that you find in directory `gms` of your Mefisto installation. Alternatively, file `clear_scale-`

.gms clears all views and adjusts the scale. To run one of these scripts, open the file (File → Merge...) and enter the relative displacement scale factor when asked for it. The displacements are scaled such that the largest value equals the relative displacement scale factor times the length of the diagonal of the smallest cuboid surrounding the model.

Figure 3.2-3 shows some of the antimetric lateral modes and Figure 3.2-4 shows some of the symmetric vertical modes.

To animate the mode shapes, first use the HarmonicToTime plugin (Tools → Plugin...) to compute the deformation at a number of points in time from the mode shapes. As normal modes are real, variable `ImaginaryPart` must be blank. Variable `NumSteps` defines the number of points in time during one period. Increasing this value results in a smoother and slower animation. All other variables can be left unchanged (see Figure 3.2-5).

For each of the selected modes, the plugin creates a new view containing all time steps. You can use again file `scale.gms` or `clear_scale.gms` to automatically adjust the scale. Subsequently, select one of these new views. You can use the right and left arrow keys on the keyboard to go through the time steps. Keeping the right arrow pressed will animate the mode. Alternatively, you can use the animation buttons on the lower left side of the screen to

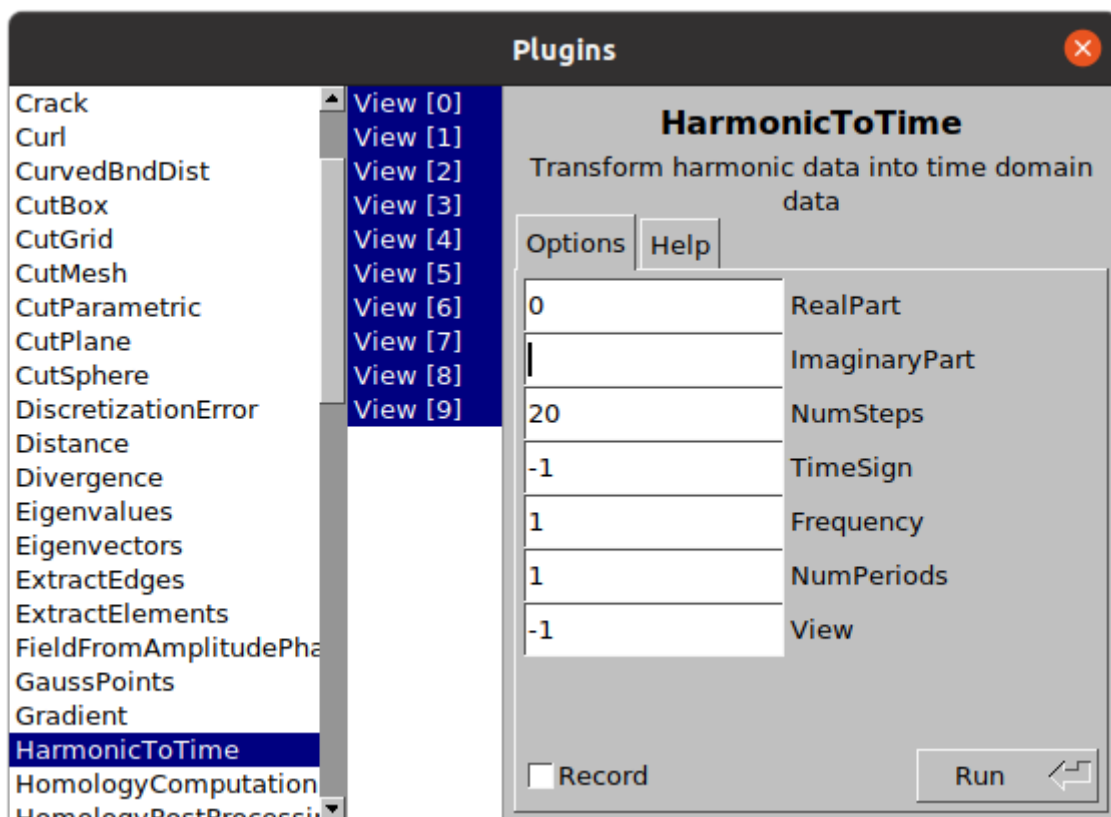


Figure 3.2-5: Gmsh: HarmonicToTime Plugin

start and stop the animation.

### 3.3 Cantilever Beam

#### Summary

Directory:	<code>exa/solid/freevib/beam3d</code>
Objectives:	<ul style="list-style-type: none"> <li>• learn how to compute and interpret the mass properties of a 3-dimensional structure</li> <li>• learn how to interpret normal modes</li> </ul>
Dimension:	3
Elements:	<b>b2, g2</b>
Constraints:	<b>rigbdy</b>
Loads:	none
Functions:	<code>mfs_line, mfs_beamsection, mfs_new, mfs_stiff, mfs_mass, mfs_massproperties, mfs_freevib, mfs_print, mfs_export</code>

#### Problem Description

Compute and interpret the mass properties and the first ten normal modes of the cantilever beam shown in Figure 3.3-1.

Data:  $a = 1000$  mm,  $b = 50$  mm,  $h = 50$  mm,  $t = 5$  mm, Young's modulus  $E = 210$  GPa, Poisson's ratio  $\nu = 0,3$ , mass density  $\rho = 7850$  kg/m<sup>3</sup>

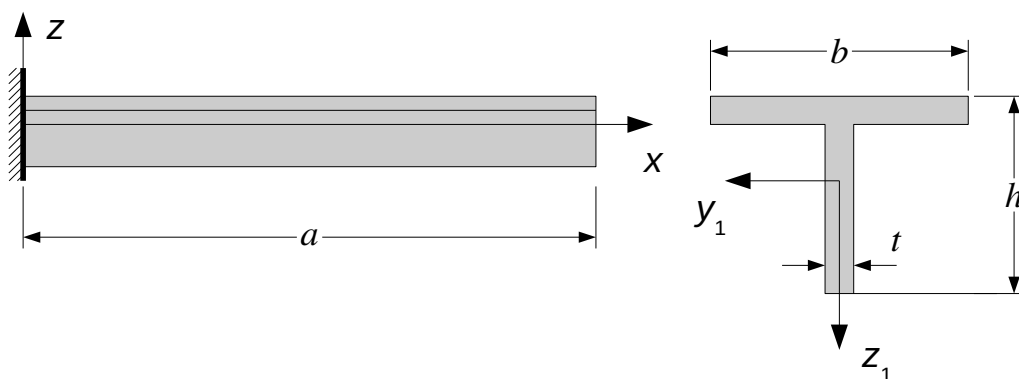


Figure 3.3-1: Cantilever Beam

## Model Definition

This simple model is defined directly in GNU Octave. To visualize the torsion, two graphical elements and rigid vertical beams are added which have no effect on the normal modes of the cantilever beam.

File `beam.m` begins with the model definition:

```
fid = fopen("beam.res", "wt");

# Data (N, mm)

a =      1000; % Length of beam
b =       50; % Width of cross section
h =       50; % Height of cross section
t =        5; % Thickness of cross section

E  =    2.1E5; % Young's modulus
ny =     0.3; % Poisson's ratio
rho = 7.85e-9; % Mass density

nel      =   20; % Number of elements
nmodes   =   10; % Number of modes

# Model definition
# -----

model = struct("type", "solid", "subtype", "3d");

# Elements along beam

geom  = mfs_beamsection("T", b, h, t);
geom.v = [0, 0, -1];

mat = struct("type", "iso", "E", E, "ny", ny, "rho", rho);

idA = 1;
idB = 1 + nel;

nodes = struct("id", {idA, idB},
               "coor", {[0, 0, 0], [a, 0, 0]});

[nodes, elem1] = mfs_line(nodes, idA, idB,
                          (idA + 1) : (idB - 1),
                          1 : nel, "b2", geom, mat);

# Graphic elements to visualize torsion

ixC = 1 + floor(nel/2);
idC = nodes(ixC).id;

idB1 = idB + 1; idC1 = idB1 + 1;

d = [0, 0, 0.1 * a];
```

```

nodes(idB1).id = idB1; nodes(idB1).coor = nodes(2).coor + d;
nodes(idC1).id = idC1; nodes(idC1).coor = nodes(ixC).coor + d;

elem(nel + 1 : nel + 2) = ...
    struct("id",      {nel + 1, nel + 2},
          "type",    "g2",
          "nodes",   {[idB, idB1], [idC, idC1]},
          "geom",    [], "mat", []);

model.nodes      = nodes;
model.elements   = elem;

# Rigid beams to connect graphic elements

model.constraints.rigbdy = struct("noda", {idB, idC},
                                "nodd", {idB1, idC1},
                                "dofs", 1 : 3);

# Constraints

# Constraints

model.constraints.prescribed = struct("id", idA,
                                     "dofs", 1 : 6);

```

Figure 3.3-2 shows the finite element mesh and the z-axis of the element system.

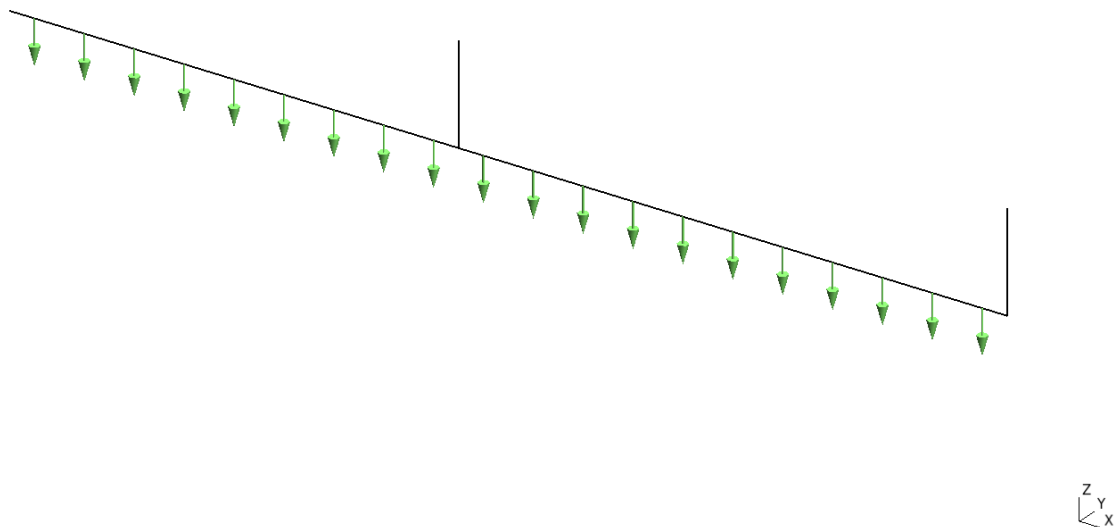


Figure 3.3-2: Beam, Finite Element Mesh and z-Axis

## Analysis

The commands to run the analysis are basically the same as those used in Example 3.2.

```
# Analysis
# -----

# Create and plot component

beam = mfs_new(fid, model);
mfs_export("beam.msh", "msh", beam, "mesh", "mesh", "axes");

# Analysis

beam = mfs_stiff(beam);
beam = mfs_mass(beam);
mfs_massproperties(fid, beam);

beam = mfs_freevib(beam, nmodes);
mfs_print(fid, beam, "modes", "freq");
mfs_export("beam.pos", "msh", beam, "modes", "disp");

fclose(fid);
```

Function **mfs\_massproperties** writes the following output to file **beam.res**:

```
Mass properties of component "beam"

Coordinates of reference point:      0.0000,      0.0000,      0.0000

Rigid body mass matrix:

    3.7288e-03  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00
    0.0000e+00  3.7288e-03  0.0000e+00  0.0000e+00  0.0000e+00  1.8644e+00
    0.0000e+00  0.0000e+00  3.7288e-03  0.0000e+00 -1.8644e+00  0.0000e+00
    0.0000e+00  0.0000e+00  0.0000e+00  1.2957e+00  0.0000e+00  0.0000e+00
    0.0000e+00  0.0000e+00 -1.8644e+00  0.0000e+00  1.2445e+03  0.0000e+00
    0.0000e+00  1.8644e+00  0.0000e+00  0.0000e+00  0.0000e+00  1.2445e+03

Mass = 3.7288e-03

Inertia tensor with respect to reference point:

    1.2957e+00  0.0000e+00  0.0000e+00
    0.0000e+00  1.2445e+03  0.0000e+00
    0.0000e+00  0.0000e+00  1.2445e+03

Coordinates of center of mass:      500.0000,      0.0000,      0.0000

Inertia tensor with respect to center of mass:

    1.2957e+00  0.0000e+00  0.0000e+00
    0.0000e+00  3.1228e+02  0.0000e+00
    0.0000e+00  0.0000e+00  3.1228e+02
```

The rigid body mass matrix is computed from

$$[m_{RR}] = [R]^T [M] [R]$$

where the columns of matrix  $[R]$  describe unit rigid body motions. The first three columns are rigid body translations in  $x$ -,  $y$ - and  $z$ -direction. The remaining three columns are rotations about the  $x$ -,  $y$ - and  $z$ -axis.

The rigid body mass matrix is composed of three 3 by 3 submatrices:

$$[m_{RR}] = \begin{bmatrix} [m_{tt}] & [m_{tr}] \\ [m_{tr}]^T & [m_{rr}] \end{bmatrix}$$

Matrix  $[m_{tt}]$  is a diagonal matrix, the diagonal elements being equal to the total

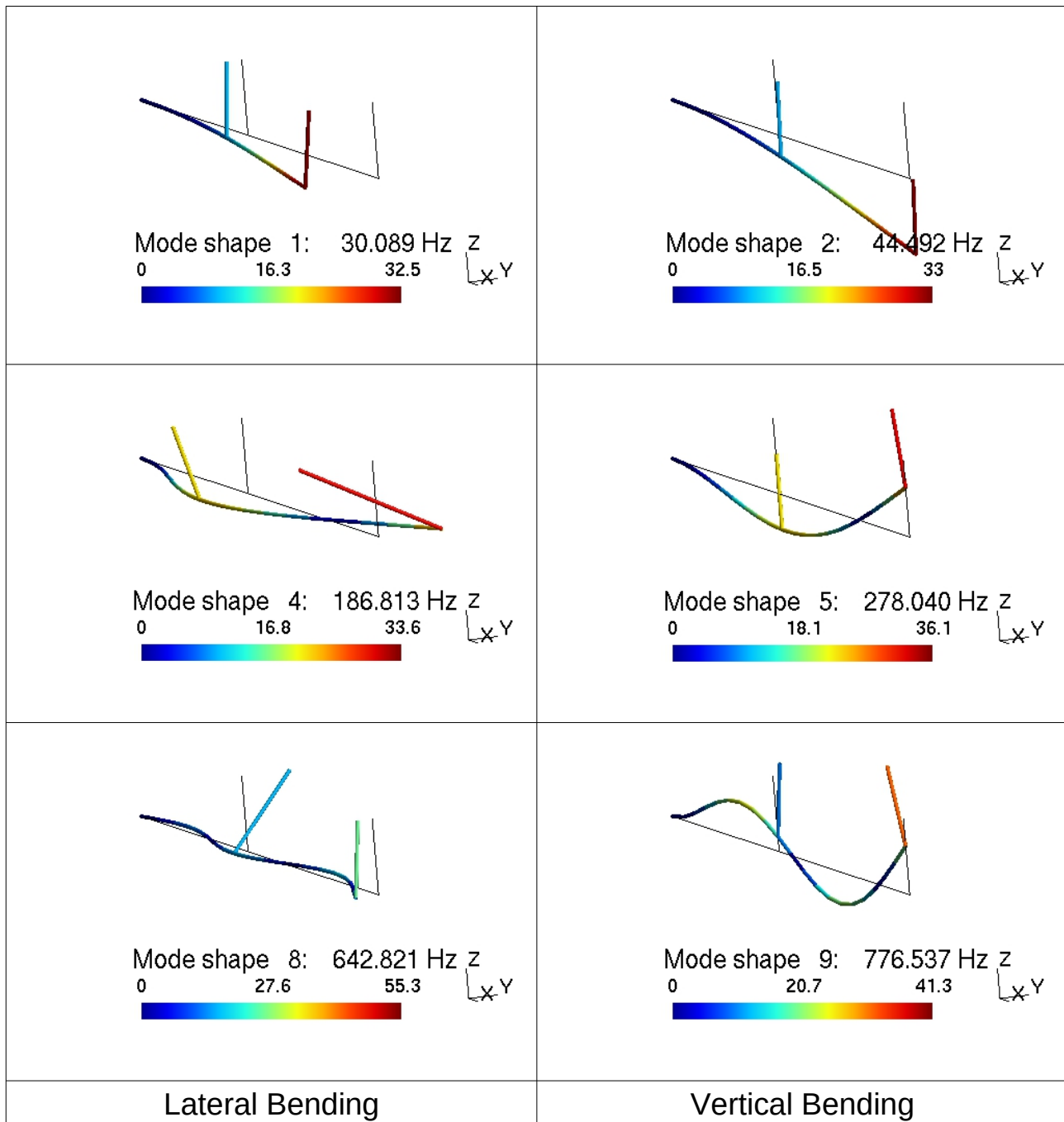


Figure 3.3-3: Beam, Some Bending Modes

mass of the structure. The symmetric matrix  $[m_{rr}]$  represents the inertia tensor with respect to the reference point. The skew-symmetric matrix  $[m_r]$  contains the static moments with respect to the reference point. When multiplied with the unit angular accelerations about the reference point, it gives the inertia forces. Its transpose, when multiplied with unit translational acceleration, gives the moments with respect to the reference point due to the inertia forces.

The output continues with the list of natural frequencies:

Component "beam"

Natural frequencies:

Mode	Circ. Frequency	Frequency
1	189.05297	30.08872 Hz
2	279.55189	44.49206 Hz
3	805.35108	128.17561 Hz
4	1173.77907	186.81274 Hz
5	1746.97553	278.03979 Hz
6	2341.87988	372.72176 Hz
7	3090.58940	491.88258 Hz
8	4038.96370	642.82104 Hz
9	4879.12500	776.53686 Hz
10	5254.83447	836.33288 Hz

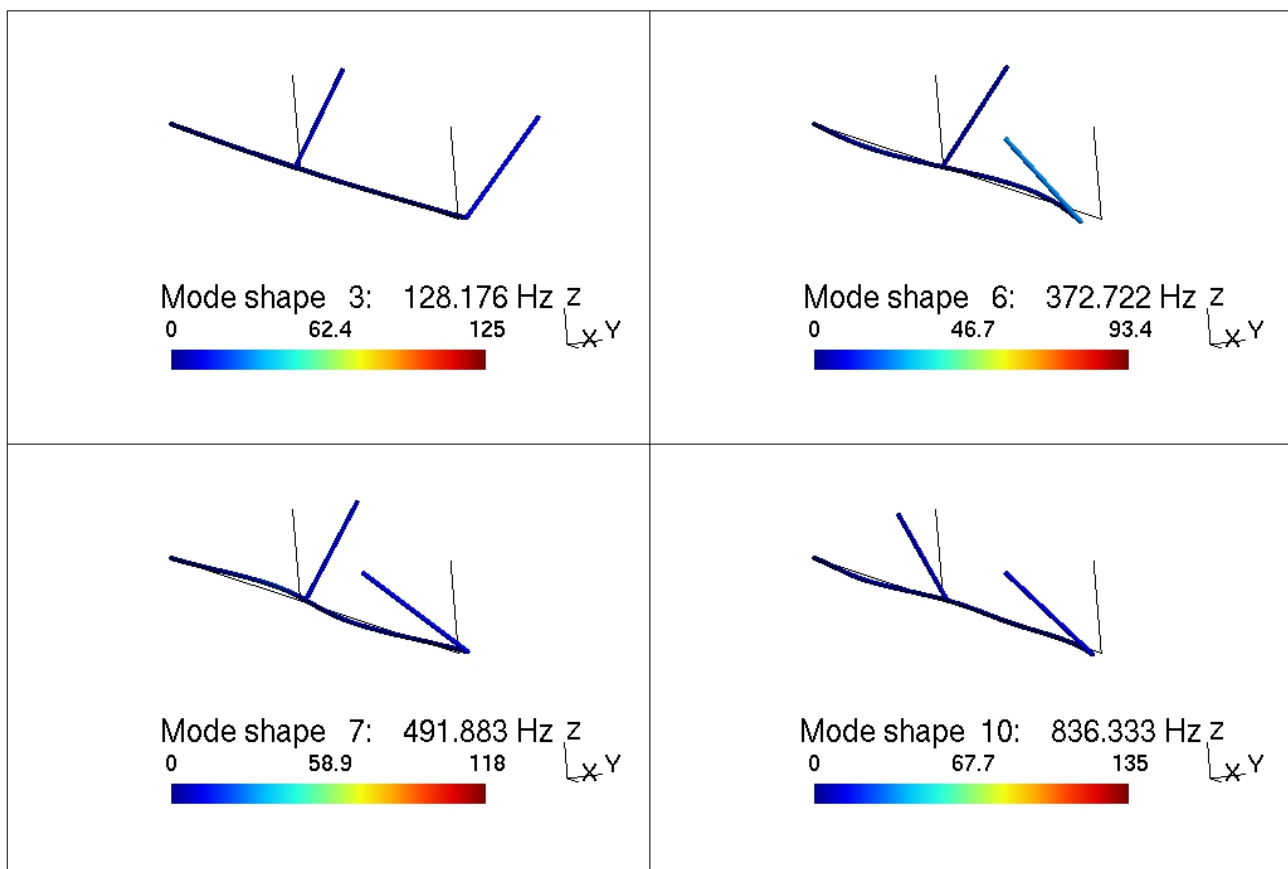


Figure 3.3-4: Beam, Some Torsional Modes



## Postprocessing

The normal modes are visualized in Gmsh. If we want to visualize the deformed structure together with the undeformed structure, we have to run the script `clear_scale.gms` by merging it instead of opening it (`File` → `Merge...`). If we open the file instead of merging it, the mesh itself can no longer be visualized.

The modes can be classified as follows:

Lateral bending	Modes 1, 4 and 8
Vertical bending	Modes 2, 5 and 9
Torsion	Modes 3, 6, 7 and 10

Figure 3.3-3 shows the bending modes and Figure 3.3-4 the torsional modes. Because the shear centre does not coincide with the centre of mass, there is a coupling between torsion and lateral bending. There is no coupling between torsion and vertical bending because both the shear centre and the centre of mass are in the  $xz$ -plane.

## 3.4 Fan Blade

### Summary

Directory:	<code>exa/solid/freevib/fanblade</code>
Objectives:	<ul style="list-style-type: none"> <li>• get familiar with shell elements in dynamic analysis</li> <li>• understand output of modal effective masses of 3-dimensional structures</li> </ul>
Dimension:	3
Elements:	<b>s4</b> , <b>s3</b>
Loads:	none
Functions:	<code>mfs_new</code> , <code>mfs_stiff</code> , <code>mfs_mass</code> , <code>mfs_massproperties</code> , <code>mfs_freevib</code> , <code>mfs_print</code> , <code>mfs_export</code> , <code>mfs_meffmass</code>

### Problem Description

Compute the normal modes of the cylindrical fan blade shown in Figure 3.4-1. Use both **s4** and **s3** elements and different mesh sizes to study the effect of

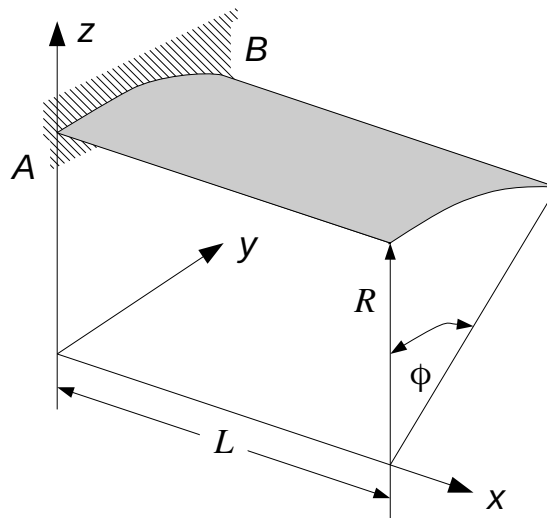


Figure 3.4-1: Fan Blade

the discretization on the results. The blade is clamped along edge  $AB$ .

Data:  $L = 304.8$  mm,  $R = 609.6$  mm, thickness  $t = 3.048$  mm,  $\phi = 0.5$  rad, Young's modulus  $E = 2.0601 \cdot 10^5$  MPa, Poisson's ratio  $\nu = 0.3$ , mass density  $\rho = 7876$  kg/m<sup>3</sup>

The data and reference results obtained with PERMAS are taken from Parisch<sup>1</sup> who also reports experimental results presented by Olson and Lindberg<sup>2</sup> who first studied this problem.

### Model Definition

File `model.m` defines the model. The definition is nearly identical with that of Example 2.7, so it is not repeated here.

### Analysis

The analysis is performed by GNU Octave script `fanblade.m`. The element type, the number of elements in longitudinal direction, the type of mass matrix and the number of modes to be computed are asked for interactively. The number of elements in circumferential direction is identical with that in longitudinal direction.

The commands to run the normal modes analysis are the same as in the previous examples, so they are not repeated here.

- 1 Parisch, H. and E. Schrem, *QUAD4 and TRIA3 Shell Elements in PERMAS*, INTES Publication UM 402, Rev. C, Stuttgart 1987
- 2 Olson, M.D. and G.M. Lindberg, *Vibration analysis of cantilevered curved plates using new cylindrical shell finite elements*, AFFDL-TR-68-150, National Aeronautical Establishment, Ottawa, Canada, 1968

Mode	1	2	3	4	5	6	7	
Measured	86.6	135.5	258.9	350.6	395.2	531.1	743.2	Hz
QUAD4	85.9	138.0	251.6	349.1	390.9	556.9	763.7	Hz
<b>s4</b>	86.2	138.5	253.1	352.9	394.9	572.4	772.9	Hz
<b>s3</b>	90.3	143.5	258.8	373.3	436.9	563.3	789.8	Hz

Table 3.4-1: Fan Blade, Comparison of Frequencies

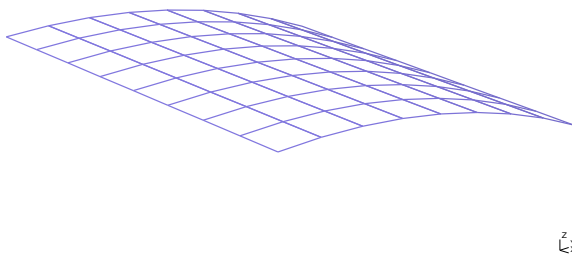
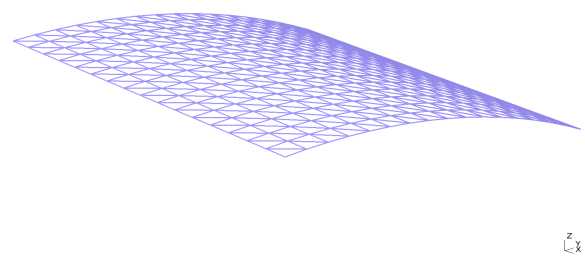
## Results

Table 3.4-1 compares the frequencies of the first seven normal modes. The measured results are the experimental results obtained by Olson and Lindberg. The QUAD4 results are obtained with 8 by 8 PERMAS QUAD4 elements, the **s4** results with 8 by 8 **s4** elements (see Figure 3.4-2) and the **s3** results with a 16 by 16 grid of **s3** elements (see Figure 3.4-3). It can be seen that the results obtained with the **s4** element are in good agreement with the measured and the QUAD4 results. To obtain a comparable accuracy with the **s3** element, at least twice the number of elements in longitudinal and circumferential direction is needed.

All finite element results have been computed with a consistent mass matrix.

Some of the mode shapes are shown in Figure 3.4-4. It can be seen that the discretization is fine enough to represent the mode shapes but for the 6<sup>th</sup> mode shape it is marginal. There are only eight linear elements per wave length. This explains why the frequency of the 6<sup>th</sup> mode obtained with the **s3** mesh, with 16 elements per wavelength, is lower than that obtained with the **s4** mesh.

For 3-dimensional structures, there are six modal effective masses. The output file contains two tables of modal effective masses, the first containing the translational modal effective masses and the second the rotational ones.

Figure 3.4-2: Fan Blade, QUAD4 and **s4** MeshFigure 3.4-3: Fan Blade, **s3** Mesh

## Modal effective masses of component "blade"

Coordinates of reference point: 0.0000, 0.0000, 0.0000

mode	frequency	tx	ty	tz	sum tx	sum ty	sum tz
1	86.17 Hz	6.4515e-33	9.5390e-03	6.2194e-04	0.00000	0.00954	0.00062
2	138.53 Hz	8.5658e-09	3.8496e-02	5.9043e-01	0.00000	0.04803	0.59105
3	253.05 Hz	9.7444e-09	3.9088e-03	5.9951e-02	0.00000	0.05194	0.65100
4	352.94 Hz	8.6541e-30	3.7989e-03	2.4768e-04	0.00000	0.05574	0.65125
5	394.95 Hz	4.2162e-06	6.4293e-03	9.8609e-02	0.00000	0.06217	0.74986
6	572.41 Hz	4.0207e-33	3.3198e-05	2.1645e-06	0.00000	0.06220	0.74986
7	772.86 Hz	5.9748e-05	1.8384e-03	2.8196e-02	0.00006	0.06404	0.77805
mode	frequency	rx	ry	rz	sum rx	sum ry	sum rz
1	86.17 Hz	4.7447e-04	7.5864e-05	6.9151e-03	0.00047	0.00008	0.00692
2	138.53 Hz	1.6979e-27	7.3019e-02	2.8294e-02	0.00047	0.07310	0.03521
3	253.05 Hz	2.2542e-29	9.6999e-03	3.7585e-03	0.00047	0.08280	0.03897
4	352.94 Hz	2.9784e-05	3.7240e-06	3.3945e-04	0.00050	0.08280	0.03931
5	394.95 Hz	1.4824e-28	1.2747e-03	4.9391e-04	0.00050	0.08407	0.03980
6	572.41 Hz	2.1007e-04	1.1815e-06	1.0769e-04	0.00071	0.08407	0.03991
7	772.86 Hz	1.5446e-28	4.3610e-04	1.6898e-04	0.00071	0.08451	0.04008

Most of the modes have very small modal effective masses, e.g. modes 1 and 6. However, Figure 3.4-4 clearly shows that these are not local modes.

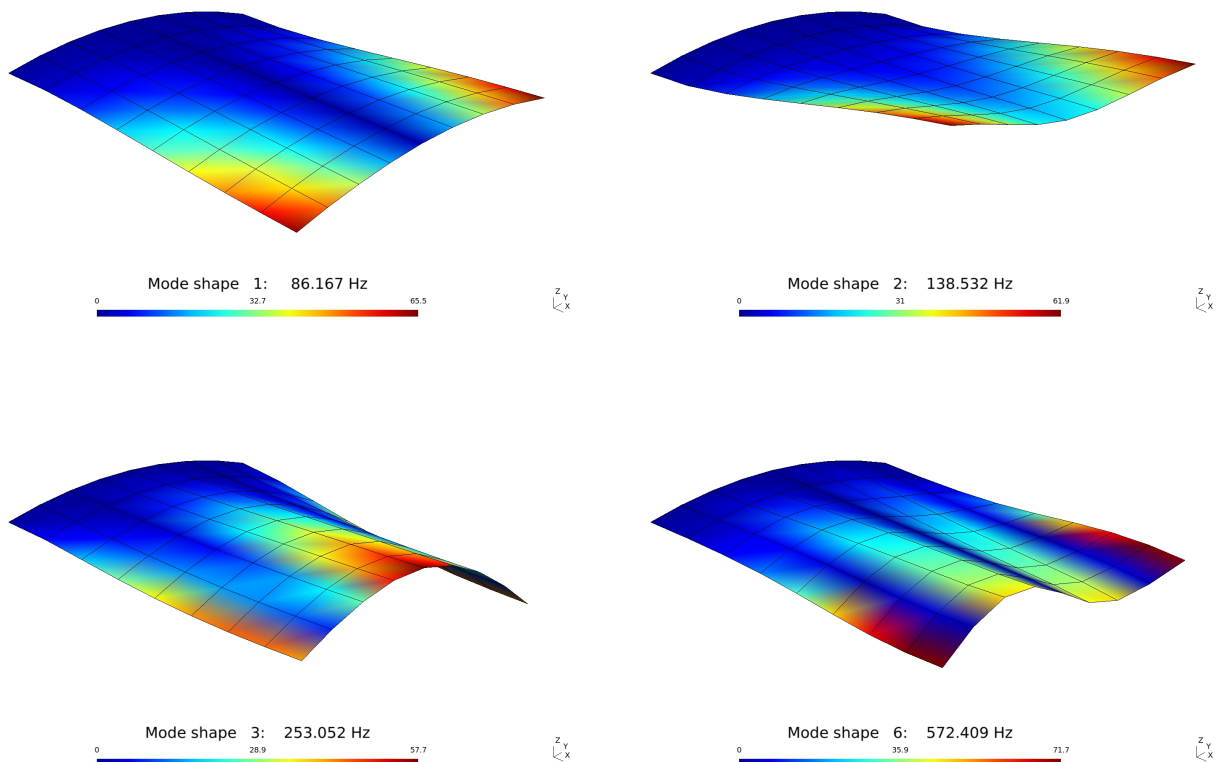


Figure 3.4-4: Fan Blade, Some Normal Modes

### 3.5 Stiffened Plate with Point Mass

#### Summary

Directory:	exa/solid/freevib/stiffened_plate
Objectives:	<ul style="list-style-type: none"> <li>• learn how to attach a point mass to a structure without stiffening the structure</li> <li>• understand the difference between <b>rigfit</b> and <b>rigbdy</b> constraints</li> <li>• learn how to interpret reaction loads of normal modes</li> <li>• understand the relationship between reaction loads and modal effective masses</li> </ul>
Dimension:	3
Elements:	<b>b2</b> , <b>s4</b> , <b>m1</b> , <b>g2</b>
Constraints:	<b>rigfit</b> , <b>ribgdy</b>
Loads:	none
Functions:	<b>mfs_beamsection</b> , <b>mfs_import</b> , <b>mfs_new</b> , <b>mfs_stiff</b> , <b>mfs_mass</b> , <b>mfs_massproperties</b> , <b>mfs_freevib</b> , <b>mfs_print</b> , <b>mfs_export</b> , <b>mfs_meffmass</b> , <b>mfs_getresp</b>

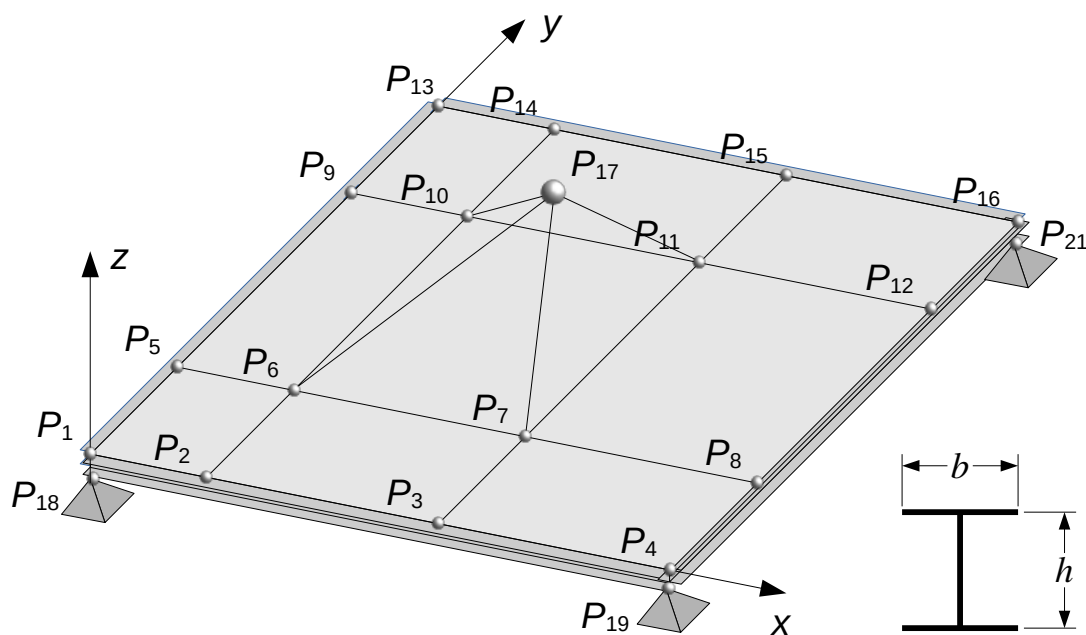


Figure 3.5-1: Stiffened Plate with Point Mass

## Problem Description

Compute the first five normal modes of the structure shown in Figure 3.5-1. Examine the reaction loads, the modal effective masses and the internal constraint loads.

The structure consists of a plate with stiffening beams attached at the lower side. The cross section of the beams is an I-section. The plate is simply supported at the four edge points  $P_{18}$ ,  $P_{19}$ ,  $P_{20}$  and  $P_{21}$ , which are located on the underside of the beams.

At points  $P_6$ ,  $P_7$ ,  $P_{10}$  and  $P_{11}$ , a device is attached to the plate, of which only the mass and the coordinates of the centre of mass are known. It is assumed that this device does not stiffen the structure. The mass of the device is 1 kg.

Coordinates:

$$x_2 = 10 \text{ cm}, x_3 = 30 \text{ cm}, x_4 = 50 \text{ cm}, x_{17} = 0.3 x_2 + 0.7 x_3$$

$$y_5 = 10 \text{ cm}, y_9 = 30 \text{ cm}, y_{13} = 40 \text{ cm}, y_{17} = 0.3 y_5 + 0.7 y_9$$

$$z_{17} = 10 \text{ cm}$$

Thickness of the plate: 2 mm

Cross section properties of the beams:

$$b = h = 10 \text{ mm}, \text{ thickness } 1 \text{ mm}$$

Material data:  $E = 70 \text{ GPa}$ ,  $\nu = 0.34$ ,  $\rho = 2700 \text{ kg/m}^3$

## Model Definition

The geometry and the mesh are defined in file `plate.geo`. As the definition of linear constraints is not supported by Gmsh, physical groups with the nodal points with dependent and autonomous degrees of freedom of the constraints are defined. The node sets resulting from these physical groups can then be used to define the linear constraints.

```
/* -----
      Stiffened plate with attached point mass
----- */

// Dimensions [mm]

DefineConstant [ x4 = 500, // Length in x-direction
                 y13 = 400, // Length in y-direction
                 x2 = 100,  // x-position of first stiffener
                 x3 = 300,  // x-position of second stiffener
                 y5 = 100,  // y-position of first stiffener
                 y9 = 300,  // y-position of second stiffener
                 z17 = 100, // z-position of mass
```

```

        zs = -10 ]; // z-position of supports

// Meshing parameters

ne = GetValue("Number of Elements along shortest edge?", 5);
Mesh.RecombineAll = 1;

// Points of plate and stiffeners

Point( 1) = { 0, 0, 0}; // Points along y = 0
Point( 2) = {x2, 0, 0};
Point( 3) = {x3, 0, 0};
Point( 4) = {x4, 0, 0};

Point( 5) = { 0, y5, 0}; // Points along y = y5
Point( 6) = {x2, y5, 0};
Point( 7) = {x3, y5, 0};
Point( 8) = {x4, y5, 0};

Point( 9) = { 0, y9, 0}; // Points along y = y9
Point(10) = {x2, y9, 0};
Point(11) = {x3, y9, 0};
Point(12) = {x4, y9, 0};

Point(13) = { 0, y13, 0}; // Points along y = y13
Point(14) = {x2, y13, 0};
Point(15) = {x3, y13, 0};
Point(16) = {x4, y13, 0};

// Point of concentrated mass

Point(17) = {0.3 * x2 + 0.7 * x3, 0.3 * y5 + 0.7 * y9, z17};

// Points of supports

Point(18) = { 0, 0, zs}; // below Point 1
Point(19) = {x4, 0, zs}; // below Point 4
Point(20) = { 0, y13, zs}; // below Point 13
Point(21) = {x4, y13, zs}; // below Point 16

// Lines 1 to 12: Stiffeners parallel to x-axis

p1 = 1; lx = 0;
For m In {0 : 3}
    For n In {0 : 2}
        lx += 1; Line(lx) = {p1++, p1};
    EndFor
    p1++;
EndFor

// Lines 13 to 24: Stiffeners parallel to y-axis

ly = lx;
For m In {0 : 3}
    p1 = 1 + m;

```

```

        For n In {0 : 2}
            p2 = p1 + 4;
            ly += 1; Line(ly) = {p1, p2};
            p1 = p2;
        EndFor
    EndFor

    Physical Curve("Stiffeners") = {1 : ly};

    Transfinite Curve {1 : 10 : 3} = ne;
    Transfinite Curve {13 : 22 : 3, 15 : 24 : 3} = ne;
    Transfinite Curve {2 : 11 : 3, 3 : 12 : 3} = 2 * ne;
    Transfinite Curve {14 : 23 : 3} = 2 * ne;

// Concentrated mass at point 17

    Physical Point("Point_Mass") = {17};

// Supports at edge nodes

    Physical Point("Supports") = {18 : 21};

// Rigfit constraint to attach mass

    // Lines to visualize constraint

    P1[0] = 6; P1[1] = 7; P1[2] = 10; P1[3] = 11;
    For k In {0 : 3}
        lno[k] = newl; Line(lno[k]) = {P1[k], 17};
    EndFor

    Physical Curve("Rigfit") = { lno[] };
    Transfinite Curve { lno[] } = 1;

    // Dependent and autonomous nodal points

    Physical Point("Rigfit_d") = 17;
    Physical Point("Rigfit_a") = {6, 7, 10, 11};

// Rigbdy constraints for supports

    // Lines to visualize constraints

    P1 = 18;
    P2[0] = 1; P2[1] = 4; P2[2] = 13; P2[3] = 16;
    For k In {0 : 3}
        lno[k] = newl; Line(lno[k]) = {P1++, P2[k]};
    EndFor

    Physical Curve("Rigbdy") = { lno[] };
    Transfinite Curve { lno[] } = 1;

    // Dependent and autonomous nodal points

    Physical Point("Rbar_1d") = 1;

```



```

Physical Point("Rbar_1a") = 18;
Physical Point("Rbar_2d") = 4;
Physical Point("Rbar_2a") = 19;
Physical Point("Rbar_3d") = 13;
Physical Point("Rbar_3a") = 20;
Physical Point("Rbar_4d") = 16;
Physical Point("Rbar_4a") = 21;

// Surfaces

lopno = 1;
l1 = 1; l2 = 16; l3 = 4; l4 = 13;

For m In {0 : 2}
    l2 = 16 + m; l3 = l1 + 3; l4 = l2 - 3;
    For n In {0 : 2}
        Curve Loop(lopno) = {l1++, l2, -l3, -l4};
        Plane Surface(news) = {lopno++};
        l4 = l2; l2 += 3; l3++;
    EndFor
EndFor

Physical Surface("Plate") = {Surface{:}};
Transfinite Surface {Surface{:}};

```

File `modes.m` begins with the translation data. First, the data, the elements and the supports are defined.

```

# Example: Stiffened plate with attached point mass
#
# -----

fid = fopen("modes.res", "wt");

# Data (N, mm)

nofmod = 5;                % Number of normal modes

geommm = struct("m", 1e-3); % Point mass
geoms = struct("t", 2);    % Shell thickness
I      = struct("b", 10, "h", 10, "t", 1, "s", 1);

mat = struct("type", "iso", "E", 70000, "ny", 0.34,
            "rho", 2.7e-9);

# Model definition

data = struct("type", "solid", "subtype", "3d");

geomb = mfs_beamsection("I", I.b, I.h, I.t, I.s);
geomb.v = [0, 0, 1];
geomb.P = [0, 0.5 * I.h];

data.Stiffeners = struct("type", "elements", "name", "b2",

```

```

        "geom", geomb, "mat", mat);

data.Plate = struct("type", "elements", "name", "s4",
                   "geom", geoms, "mat", mat);

data.Point_Mass = struct("type", "elements", "name", "m1",
                        "geom", geommm);

data.Supports = struct("type", "constraints",
                      "name", "prescribed", "dofs", 1 : 3);

```

Next, the data required to define the linear constraints are specified. Physical groups **Rigfit** and **Rigbdy** contain the graphical elements used to visualize the constraints. These elements are not necessary for the analysis, but it is good practice to use graphical elements to control the correct definition of the constraints. The remaining physical groups contain the nodal points with dependent and autonomous degrees of freedom.

```

data.Rigfit = struct("type", "elements", "name", "g2");

data.Rigfit_d = struct("type", "nodeset"); % dependent
data.Rigfit_a = struct("type", "nodeset"); % autonomous

data.Rigbdy = struct("type", "elements", "name", "g2");

data.Rbar_1d = struct("type", "nodeset");
data.Rbar_1a = struct("type", "nodeset");
data.Rbar_2d = struct("type", "nodeset");
data.Rbar_2a = struct("type", "nodeset");
data.Rbar_3d = struct("type", "nodeset");
data.Rbar_3a = struct("type", "nodeset");
data.Rbar_4d = struct("type", "nodeset");
data.Rbar_4a = struct("type", "nodeset");

[model, nsets] = mfs_import(fid, "plate.msh", "msh", data);

```

Function **mfs\_import** returns structure **model** with the model definition and structure **nsets** with the nodal point sets. Now, the definition of the linear constraints is added to structure **model**. Field **nodd** of the definition of the **rigfit** constraint contains the identifier of the nodal point with the dependent degrees of freedom. Field **dofa** contains a cell array with pairs of lists of nodal point identifiers and degree of freedom identifiers that define the autonomous degrees of freedom. This cell array must be enclosed by additional curly brackets to prevent GNU Octave from creating a structure array.

#### # Linear Constraints

```

model.constraints.rigfit = ...
    struct("nodd", nsets.Rigfit_d,
          "dofa", {{nsets.Rigfit_a, 1 : 3}});

```

The **rigfit** constraint is used to attach the point mass to the structure. The displacements at the six dependent degrees of freedom of the specified nodal point are determined such that the resulting rigid body motion best matches the displacements at the autonomous degrees of freedom.

In contrast, the **rigbdy** constraint defines a rigid body. The displacements at the dependent degrees of freedom follow a rigid body motion defined by the six autonomous degrees of freedom. We have to define four such constraints, each connecting one edge point of the plate to the corresponding support. Because the displacements at the supports are prescribed, the degrees of freedom at the support points must be autonomous. Dependent degrees of freedom cannot be prescribed.

The four constraints are defined with a structure array. The degrees of freedom connected to the rigid body are the same for all four constraints but the nodal points are different. Cell array **Rbar\_a** contains the identifiers of the autonomous nodal points and cell array **Rbar\_d** the identifiers of the dependent nodal points. The subsequent **struct** command creates the required structure array.

```
Rbar_a = {nsets.Rbar_1a, nsets.Rbar_2a, ...  
          nsets.Rbar_3a, nsets.Rbar_4a};  
Rbar_d = {nsets.Rbar_1d, nsets.Rbar_2d, ...  
          nsets.Rbar_3d, nsets.Rbar_4d};  
model.constraints.rigbdy = ...  
    struct("dofs", 1 : 6,  
          "noda", Rbar_a,  
          "nodd", Rbar_d);
```

## Analysis

The commands to run the normal modes analysis are the same as in the previous examples. Printed output includes the frequencies, the reaction loads and the internal constraint loads (item **ics1**). The internal constraint loads are the loads at the dependent and the autonomous degrees of freedom. These loads must be self-equilibrating.

```
# Analysis  
  
plate = mfs_new(fid, model);  
  
plate = mfs_stiff(plate);  
plate = mfs_mass(plate);  
mfs_massproperties(fid, plate);  
  
plate = mfs_freevib(plate, nofmod);
```

```
mfs_print(fid, plate, "modes", "freq", "reac", "icsl");
mfs_export("modes.dsp", "msh", plate, "modes", "disp");
```

Subsequently, the modal effective masses are computed. The modal participation factors  $\Gamma_{kn}$  are equal to the square root of the absolute modal effective masses. The absolute values of the reaction load resultants are obtained from

$$|R_{kn}| = \omega_n^2 \Gamma_{kn}$$

where  $\omega_n$  are the circular eigenfrequencies.

```
mem = mfs_meffmass(fid, plate);
mpf = sqrt([mem.abs]);
w = 2 * pi * mfs_getresp(plate, "modes", "freq");
R = mpf .* w.^2;

fprintf(fid,
        "Reaction load resultants from effective masses\n\n");
fdisp(fid, R);

fclose(fid);
```

## Results

The following results were obtained with the proposed default value for the number of elements. First, let us look at the reaction loads:

-----

Component "plate"

Natural frequencies:

Mode	Circ. Frequency	Frequency
1	351.66949	55.96994 Hz
2	697.25889	110.97220 Hz
3	863.00887	137.35213 Hz
4	1385.81357	220.55908 Hz
5	1615.30070	257.08309 Hz

Reaction loads of mode 1

node	Fx	Fy	Fz	Mx	My	Mz
18	-7.352e+03	-3.217e+03	-1.053e+03	0.000e+00	0.000e+00	0.000e+00
19	7.911e+03	-5.215e+03	-1.257e+03	0.000e+00	0.000e+00	0.000e+00
20	-1.105e+04	3.533e+03	-1.534e+03	0.000e+00	0.000e+00	0.000e+00
21	1.120e+04	5.077e+03	-1.564e+03	0.000e+00	0.000e+00	0.000e+00
Res.	6.996e+02	1.776e+02	-5.408e+03	-1.238e+06	1.404e+06	-1.253e+05

Reaction loads of mode 2

node	Fx	Fy	Fz	Mx	My	Mz
18	-3.070e+04	-3.005e+03	-3.633e+03	0.000e+00	0.000e+00	0.000e+00
19	2.933e+04	-4.233e+03	-3.309e+03	0.000e+00	0.000e+00	0.000e+00
20	2.630e+04	-3.583e+03	3.338e+03	0.000e+00	0.000e+00	0.000e+00

21	-2.391e+04	9.131e+02	2.220e+03	0.000e+00	0.000e+00	0.000e+00
Res.	1.031e+03	-9.908e+03	-1.384e+03	2.124e+06	5.341e+05	-2.618e+06

Reaction loads of mode 3

node	Fx	Fy	Fz	Mx	My	Mz
18	8.072e+03	3.423e+04	5.721e+03	0.000e+00	0.000e+00	0.000e+00
19	-1.183e+03	-1.962e+04	-2.527e+03	0.000e+00	0.000e+00	0.000e+00
20	5.663e+03	-3.304e+04	5.359e+03	0.000e+00	0.000e+00	0.000e+00
21	2.307e+03	1.918e+04	-2.834e+03	0.000e+00	0.000e+00	0.000e+00
Res.	1.486e+04	7.545e+02	5.719e+03	1.018e+06	2.532e+06	-3.408e+06

Reaction loads of mode 4

node	Fx	Fy	Fz	Mx	My	Mz
18	2.056e+04	-2.606e+04	-1.433e+03	0.000e+00	0.000e+00	0.000e+00
19	-2.749e+04	-5.542e+04	-3.563e+03	0.000e+00	0.000e+00	0.000e+00
20	1.810e+04	2.560e+04	-1.564e+03	0.000e+00	0.000e+00	0.000e+00
21	-2.554e+04	5.676e+04	-3.891e+03	0.000e+00	0.000e+00	0.000e+00
Res.	-1.437e+04	8.837e+02	-1.045e+04	-2.173e+06	3.871e+06	3.649e+06

Reaction loads of mode 5

node	Fx	Fy	Fz	Mx	My	Mz
18	5.099e+04	2.868e+04	8.940e+03	0.000e+00	0.000e+00	0.000e+00
19	-4.441e+04	3.450e+04	8.283e+03	0.000e+00	0.000e+00	0.000e+00
20	5.076e+04	-2.795e+04	8.915e+03	0.000e+00	0.000e+00	0.000e+00
21	-4.254e+04	-3.496e+04	8.110e+03	0.000e+00	0.000e+00	0.000e+00
Res.	1.481e+04	2.698e+02	3.425e+04	6.813e+06	-8.345e+06	-3.519e+06

The largest resultant force of mode 1 is in z-direction, of mode 2 in y-direction and of mode 3 in x-direction. Of modes 4 and 5, the predominant forces are in x- and z-direction. This indicates that the motion of mode 1 is mainly in vertical direction, of mode 2 in lateral direction and of mode 3 in longitudinal direction whereas modes 4 and 5 have in motion in both longitudinal and vertical direction. Figure 3.5-2 confirms these conclusions.

The output continues with the internal constraint loads. Only the internal constraint loads of the first mode are shown here.

Internal constraint loads of mode 1

node	Fx	Fy	Fz	Mx	My	Mz
1	-7.352e+03	-3.217e+03	-1.053e+03	-3.217e+04	7.352e+04	-3.638e-12
4	7.911e+03	-5.215e+03	-1.257e+03	-5.215e+04	-7.911e+04	0.000e+00
6	-1.484e+02	-7.169e+01	-7.669e+01	0.000e+00	0.000e+00	0.000e+00
7	-1.484e+02	-2.053e+01	8.450e+02	0.000e+00	0.000e+00	0.000e+00
10	-1.996e+02	-7.169e+01	5.893e+02	0.000e+00	0.000e+00	0.000e+00
11	-1.996e+02	-2.053e+01	1.511e+03	0.000e+00	0.000e+00	0.000e+00
13	-1.105e+04	3.533e+03	-1.534e+03	3.533e+04	1.105e+05	5.457e-12
16	1.120e+04	5.077e+03	-1.564e+03	5.077e+04	-1.120e+05	3.638e-12
17	6.960e+02	1.844e+02	-2.869e+03	0.000e+00	0.000e+00	0.000e+00
18	7.352e+03	3.217e+03	1.053e+03	0.000e+00	0.000e+00	0.000e+00
19	-7.911e+03	5.215e+03	1.257e+03	0.000e+00	0.000e+00	0.000e+00
20	1.105e+04	-3.533e+03	1.534e+03	0.000e+00	0.000e+00	0.000e+00
21	-1.120e+04	-5.077e+03	1.564e+03	0.000e+00	0.000e+00	0.000e+00
Res.	7.458e-11	-1.819e-12	2.800e-09	2.624e-08	-7.618e-07	-2.374e-08

... ..

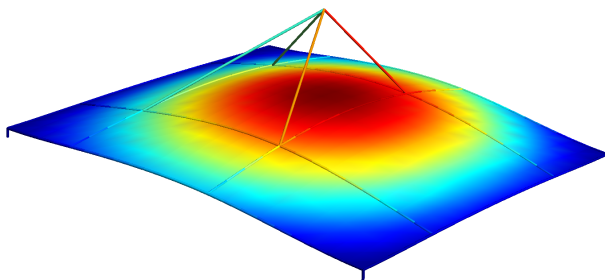
The point mass is attached to nodal point 17 which is connected to nodal points 6, 7, 10 and 11 by the **rigfit** constraint. It can be seen how the inertia loads of the point mass (in red) are transmitted to the nodal points of the plate (in green). The loads marked in green are in equilibrium with the loads marked in red. Furthermore, the loads at all autonomous degrees of freedom are in equilibrium with the loads at all dependent degrees of freedom, so that the resultant of the internal constraint loads is zero.

The internal constraint loads are followed by the modal effective masses:

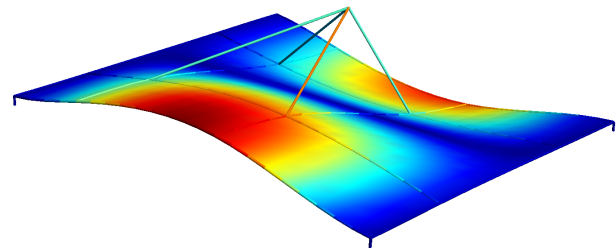
-----  
Modal effective masses of component "plate"

Coordinates of reference point: 0.0000, 0.0000, 0.0000

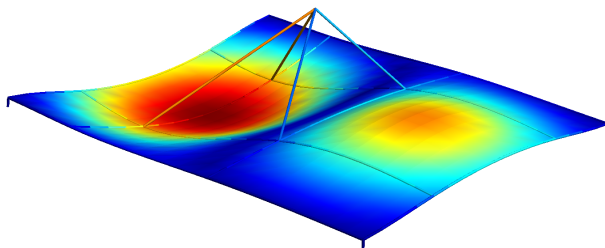
mode	frequency	tx	ty	tz	sum tx	sum ty	sum tz
1	55.97 Hz	1.3664e-02	8.8121e-04	8.1665e-01	0.01366	0.00088	0.81665
2	110.97 Hz	1.9215e-03	1.7735e-01	3.4594e-03	0.01559	0.17823	0.82011
3	137.35 Hz	1.7000e-01	4.3818e-04	2.5178e-02	0.18558	0.17867	0.84529
4	220.56 Hz	2.3899e-02	9.0365e-05	1.2647e-02	0.20948	0.17876	0.85793
5	257.08 Hz	1.3769e-02	4.5625e-06	7.3575e-02	0.22325	0.17876	0.93151



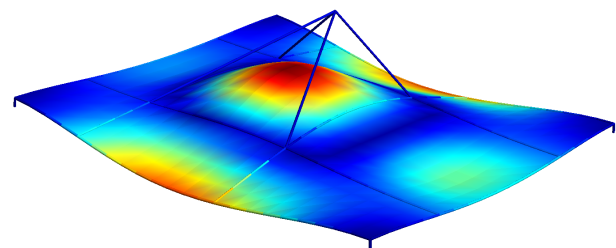
Mode 1: 55.97 Hz



Mode 2: 110.97 Hz



Mode 3: 137.4 Hz



Mode 5: 257.1 Hz

Figure 3.5-2: Stiffened Plate, Some Normal Modes

mode	frequency	rx	ry	rz	sum rx	sum ry	sum rz
1	55.97 Hz	7.1135e-01	7.1706e-01	3.4161e-03	0.71135	0.71706	0.00342
2	110.97 Hz	1.3563e-01	6.7190e-03	9.6534e-02	0.84698	0.72378	0.09995
3	137.35 Hz	1.3266e-02	6.4332e-02	6.9729e-02	0.86024	0.78811	0.16968
4	220.56 Hz	9.0967e-03	2.2615e-02	1.2021e-02	0.86934	0.81073	0.18170
5	257.08 Hz	4.8433e-02	5.6940e-02	6.0577e-03	0.91777	0.86767	0.18776

The modal effective masses confirm the conclusions drawn from the reaction load resultants.

The output ends with the reaction loads computed from the modal effective masses:

Reaction load resultants from effective masses						
6.9955e+02	1.7765e+02	5.4081e+03	1.2375e+06	1.4036e+06	1.2528e+05	
1.0313e+03	9.9074e+03	1.3837e+03	2.1243e+06	5.3411e+05	2.6179e+06	
1.4860e+04	7.5443e+02	5.7188e+03	1.0178e+06	2.5318e+06	3.4085e+06	
1.4367e+04	8.8342e+02	1.0451e+04	2.1731e+06	3.8708e+06	3.6493e+06	
1.4815e+04	2.6969e+02	3.4248e+04	6.8127e+06	8.3447e+06	3.5196e+06	

Comparing these values with the absolute values of the reaction load resultants, we can observe a good agreement. The small differences are due the fact that the contributions from the prescribed degrees of freedom to the mass matrix are neglected when computing the modal effective masses.

### 3.6 Cylindrical Box

#### Summary

Directory:	exa/solid/freevib/cylinder
Objectives:	<ul style="list-style-type: none"> <li>• learn how to define the model of a more realistic structure</li> <li>• learn how to run a normal modes analysis of a structure with rigid body modes</li> <li>• understand normal modes of curved shell structures and of composed structures</li> </ul>
Dimension:	3
Elements:	<b>b2</b> , <b>s4</b>
Constraints:	none
Loads:	none
Functions:	<b>mfs_beamsection</b> , <b>mfs_import</b> , <b>mfs_new</b> , <b>mfs_stiff</b> , <b>mfs_mass</b> , <b>mfs_massproperties</b> , <b>mfs_freevib</b> , <b>mfs_print</b> , <b>mfs_export</b>

### Problem Description

Compute the first 20 normal modes of the stiffened cylindrical box shown in Figure 3.6-1. The box is not supported. It can be regarded as a highly simplified model of a space station.

The model consists of a cylindrical shell closed by a left and a right cap and a floor. The cylindrical shell is stiffened by frames and stringers that are attached to the inner side of the shell. The caps are stiffened by beams attached to the outer side. The floor is supported by beams attached to the lower side. It is only connected to the cylindrical shell, but not to the caps.

The model is symmetric with respect to the  $xz$ -plane. Figure 3.6-1 shows only half of the model so that the floor is also visible.

The cylinder has a length of  $L = 10$  m and a radius of  $r = 2$  m. The cross section of the frames and stringers is an I-section with a height and width of 20 mm and a thickness of 2 mm. The cross section of the stiffeners of the cap is an I-section with a height and width of 50 mm and a thickness of 5 mm. The cross section of the beams supporting the floor is a thin-walled box with a height of 100 mm, a width of 50 mm and a thickness of 5 mm.

The material is aluminium with a Young's modulus of 70 GPa, a Poisson's ra-

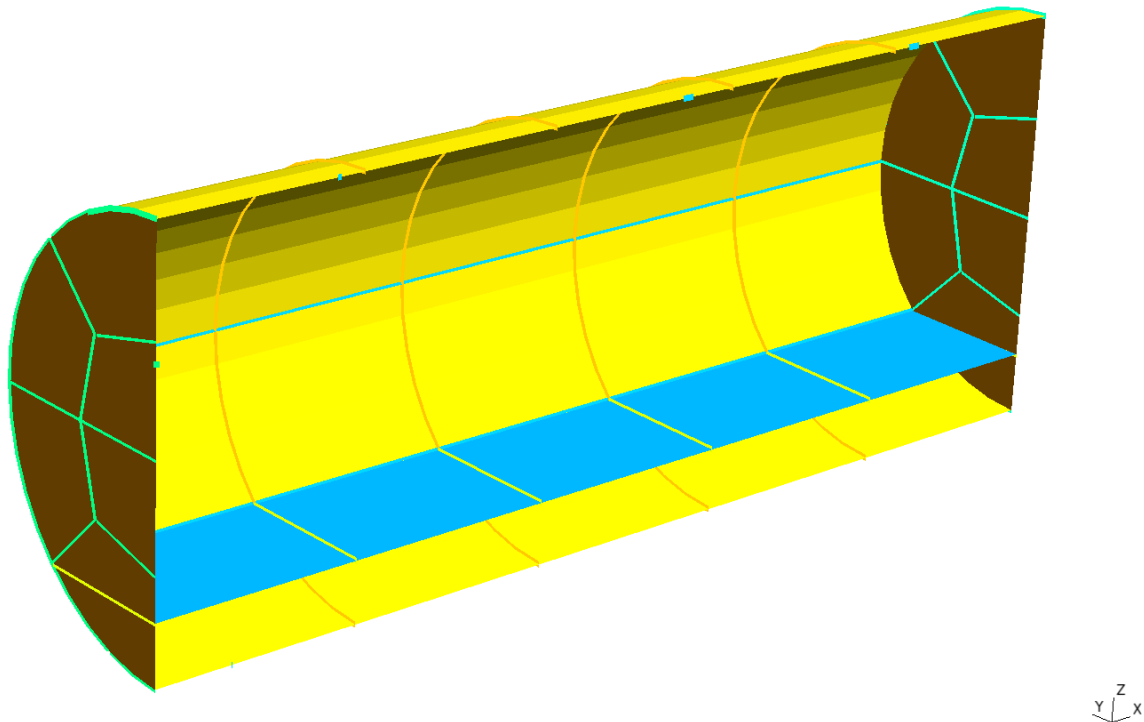


Figure 3.6-1: Cylindrical Box



tio of 0.34 and a mass density of  $2700 \text{ kg/m}^3$ .

### Model Definition

The geometry and the mesh are defined in file `cylinder.geo`. This file is quite extensive, so it is not included here. It is similar to that of Example 3.5 but in addition uses circles, rotations and translations.

The geometry comprises the following physical groups:

- **Skin**: shell elements of the cylindrical shell
- **Caps**: shell elements of the left and right cap
- **Floor**: shell elements of the floor
- **Frames**: beam elements of the frames
- **Stringers**: beam elements of the stringers
- **Left\_Stiffeners**: beam elements stiffening the left cap
- **Right\_Stiffeners**: beam elements stiffening the right cap
- **Floor\_Bars**: beam elements supporting the floor.

The translation data and the parameters controlling the analysis are defined in file `modes.m`. First, the number of requested normal modes and the options controlling the computation of the normal modes are defined.

The options controlling the computation of the normal modes are defined with structure `opts` that is used as input to function `mfs_freevib`. Only those parameters whose values deviate from the default values must be specified. With this model, it turns out that the default value of 6 for the rigid body criterion (parameter `rbc`) is not sufficient to automatically detect all rigid body modes. The value is therefore increased to 8. Alternatively, we could use parameter `rdofs` to define a set of statically determinate supports.

```
# Example: Stiffened cylindrical box
#
# -----

fid = fopen("modes.res", "wt");

# Data (N, mm)

% Parameters

nofmod = 20; % Number of normal modes
opts.rbc = 8; % Criterion to detect rigid body modes
```

Next, the shell thicknesses, the cross section properties and the material data

are defined.

```
% Shell thicknesses

ends  = struct("t", 5); % Shell thickness of ends
skin  = struct("t", 2); % Shell thickness of skin
floor = struct("t", 5); % Shell thickness of floor

% Cross section of frames and stringers

S1.h = 20; % Height of I-section
S1.b = 20; % Width of I-section
S1.t = 2;  % Thickness of I-section

frame    = mfs_beamsection("I", S1.b, S1.h, S1.t, S1.t, "rot");
stringer = mfs_beamsection("I", S1.b, S1.h, S1.t, S1.t);

% Cross section of stiffeners of front and end

S2.h = 50;
S2.b = 50;
S2.t = 5;

caps = mfs_beamsection("I", S2.b, S2.h, S2.t, S2.t);

% Cross section of floor bars

S3.h = 100;
S3.b = 50;
S3.t = 5;

floor_bars = mfs_beamsection("box", "thin", S3.b, S3.h, S3.t);

% Material data

mat = struct("type", "iso", "E", 70000, "ny", 0.34,
            "rho", 2.7e-9);
```

The model definition is completed by defining the translation data.

```
# Model definition

data = struct("type", "solid", "subtype", "3d");

% Beam elements

frame.v = [1, 0, 0];
frame.P = [-0.5 * S1.h, 0];
data.Frames = struct("type", "elements", "name", "b2",
                    "geom", frame, "mat", mat);

stringer.Q = [5000, 0, 0];
stringer.P = [0, -0.5 * S1.h];
data.Stringers = struct("type", "elements", "name", "b2",
                      "geom", stringer, "mat", mat);
```

```

caps.v = [-1, 0, 0];
caps.P = [0, -0.5 * S2.h];
data.Left_Stiffeners = struct("type", "elements", "name", "b2",
                              "geom", caps, "mat", mat);

caps.v = -caps.v;
data.Right_Stiffeners = struct("type", "elements", "name", "b2",
                              "geom", caps, "mat", mat);

floorBars.v = [0, 0, 1];
floorBars.P = [0, 0.5 * S3.h];
data.Floor_Bars = struct("type", "elements", "name", "b2",
                         "geom", floorBars, "mat", mat);

% Shell elements

data.Caps = struct("type", "elements", "name", "s4",
                  "geom", ends, "mat", mat);
data.Skin = struct("type", "elements", "name", "s4",
                  "geom", skin, "mat", mat);
data.Floor = struct("type", "elements", "name", "s4",
                   "geom", floor, "mat", mat);

```

## Analysis

The commands to run the normal modes analysis are the same as in the previous examples. The only difference is the additional input argument **opts** of function **mfs\_freevib**. This argument is required to change the value of parameter **rbc**.

```

# Analysis

model = mfs_import(fid, "cylinder.msh", "msh", data);
cyl = mfs_new(fid, model);
mfs_export("cylinder.axes", "msh", cyl, "mesh", "axes");

cyl = mfs_stiff(cyl);
cyl = mfs_mass(cyl);
mfs_massproperties(fid, cyl);

cyl = mfs_freevib(cyl, nofmod, opts);
mfs_print(fid, cyl, "modes", "freq");
mfs_export("modes.dsp", "msh", cyl, "modes", "disp");

fclose(fid);

```

## Results

The following results were obtained with the proposed default values for the number of elements.

File modes.res contains the following information:

Reading model from file "cylinder.msh", MSH file version 4.1

Physical Group	Type
Left_Stiffeners	elements
Caps	elements
Right_Stiffeners	elements
Frames	elements
Stringers	elements
Skin	elements
Floor	elements
Floor_Bars	elements

Mefisto 2.7: Building new component from input "model"

Model Type = solid, Model Subtype = 3d

Number of nodes = 2858, Number of elements = 3742  
 Number of element types = 2  
 Number of global degrees of freedom = 17148  
 Number of local degrees of freedom = 17148  
 Number of prescribed degrees of freedom = 0  
 Number of dependent degrees of freedom = 0

Mass properties of component "cyl"

Coordinates of reference point: 0.0000, 0.0000, 0.0000

Rigid body mass matrix:

1.6215e+00	0.0000e+00	0.0000e+00	0.0000e+00	-6.4064e+02	6.9252e-10
0.0000e+00	1.6215e+00	-5.2171e-19	6.4064e+02	2.6236e-15	8.1075e+03
0.0000e+00	4.3307e-19	1.6215e+00	-6.9253e-10	-8.1075e+03	2.1933e-15
0.0000e+00	6.4064e+02	-6.9251e-10	5.0469e+06	3.4585e-06	3.2032e+06

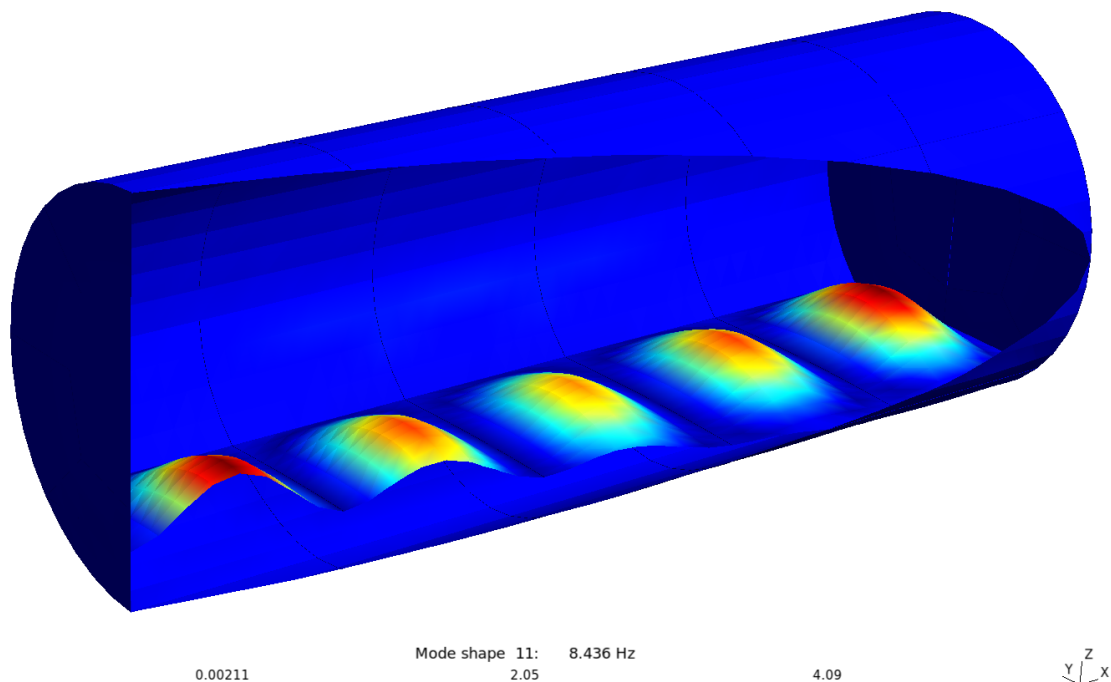


Figure 3.6-2: Cylindrical Box, Mode 11: 8.436 Hz

```

-6.4064e+02 -2.1965e-15 -8.1075e+03  3.4588e-06  6.4657e+07  4.5278e-07
 6.9253e-10  8.1075e+03 -2.6270e-15  3.2032e+06  4.5283e-07  6.4049e+07

```

Mass = 1.6215e+00

Inertia tensor with respect to reference point:

```

5.0469e+06  3.4585e-06  3.2032e+06
3.4588e-06  6.4657e+07  4.5278e-07
3.2032e+06  4.5283e-07  6.4049e+07

```

Coordinates of center of mass: 5000.0000, -0.0000, -395.0907

Inertia tensor with respect to center of mass:

```

4.7938e+06 -4.1569e-09  2.7940e-09
-3.8077e-09  2.3867e+07  7.2639e-07
2.3283e-09  7.2645e-07  2.3511e+07

```

Component "cyl"

Natural frequencies:

Mode	Circ. Frequency	Frequency
1	0.00000	0.00000 Hz
2	0.00000	0.00000 Hz
3	0.00004	0.00001 Hz
4	0.00000	0.00000 Hz
5	0.00001	0.00000 Hz
6	0.00000	0.00000 Hz
7	48.55389	7.72759 Hz
8	48.84655	7.77417 Hz
9	49.09014	7.81294 Hz
10	51.62663	8.21663 Hz
11	53.00242	8.43560 Hz

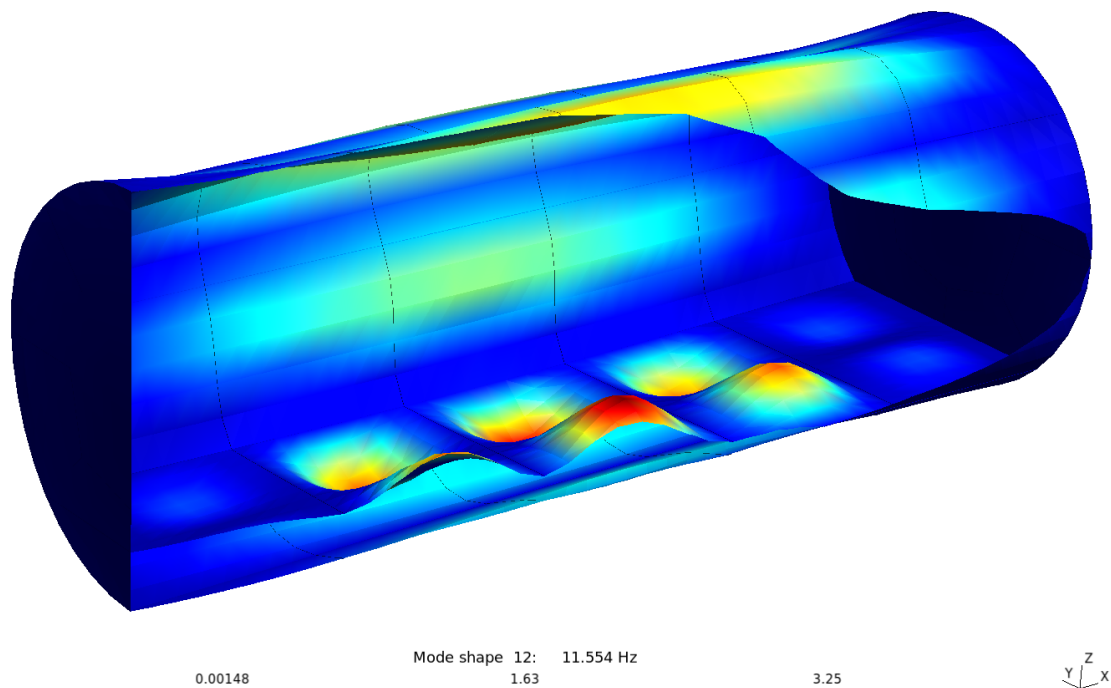


Figure 3.6-3: Cylindrical Box, Mode 12: 11.55 Hz

12	72.59680	11.55414 Hz
13	73.78940	11.74395 Hz
14	75.38360	11.99767 Hz
15	75.61031	12.03375 Hz
16	76.05451	12.10445 Hz
17	76.17279	12.12328 Hz
18	76.66245	12.20121 Hz
19	77.66789	12.36123 Hz
20	81.26244	12.93332 Hz

Because there are no constraints, the number of local degrees of freedom equals the number of global degrees of freedom.

The list of frequencies shows that the first six normal modes are rigid body modes. The frequencies are numerical zeroes.

The frequencies of the elastic modes occur in clusters, i.e. groups of frequencies that are very closer together. This is a typical phenomenon of complex 3-dimensional structures that are composed of shells.

The elastic modes can be subdivided into modes of the floor, modes of the caps, modes of the cylindrical shell and combined modes. Modes 7 to 11 and 14 to 17 are modes of the floor. As an example, Figure 3.6-2 shows mode 11. Modes 12 and 19 are combined modes of the floor and the cylindrical shell, see Figures 3.6-3 and 3.6-6. Mode 18, shown in Figure 3.6-5, is a mode of the cylindrical shell, and modes 13 and 20, shown in Figures 3.6-4 and 3.6-7, are modes of the caps.

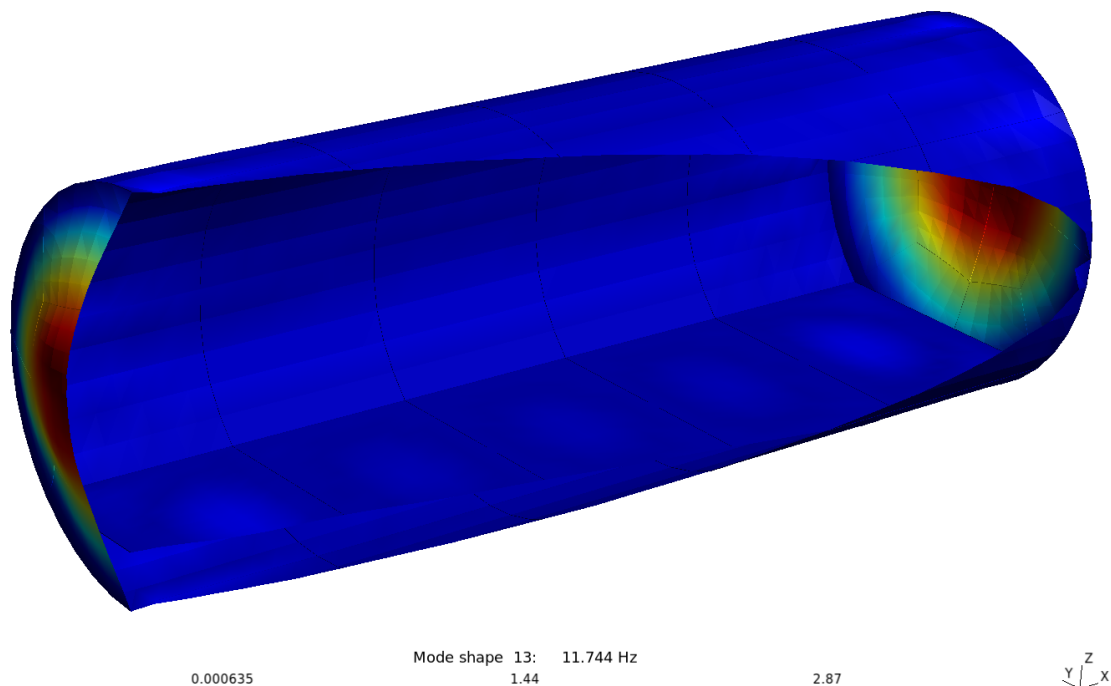


Figure 3.6-4: Cylindrical Box, Mode 13: 11.74 Hz

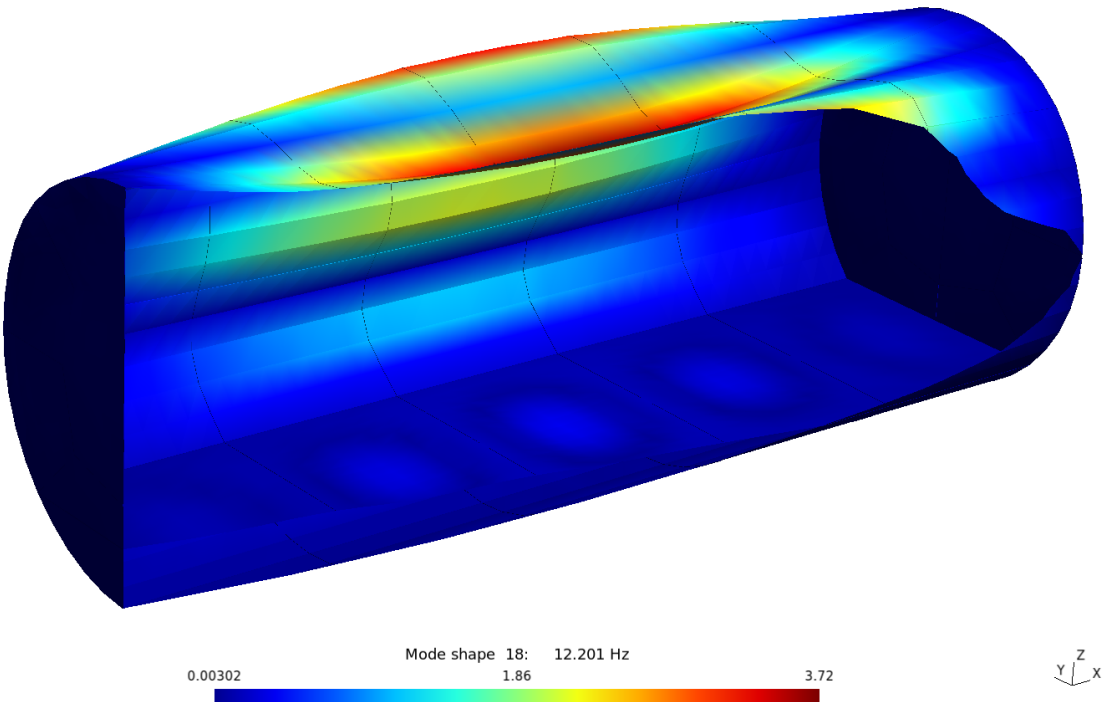


Figure 3.6-5: Cylindrical Box, Mode 18: 12.20 Hz

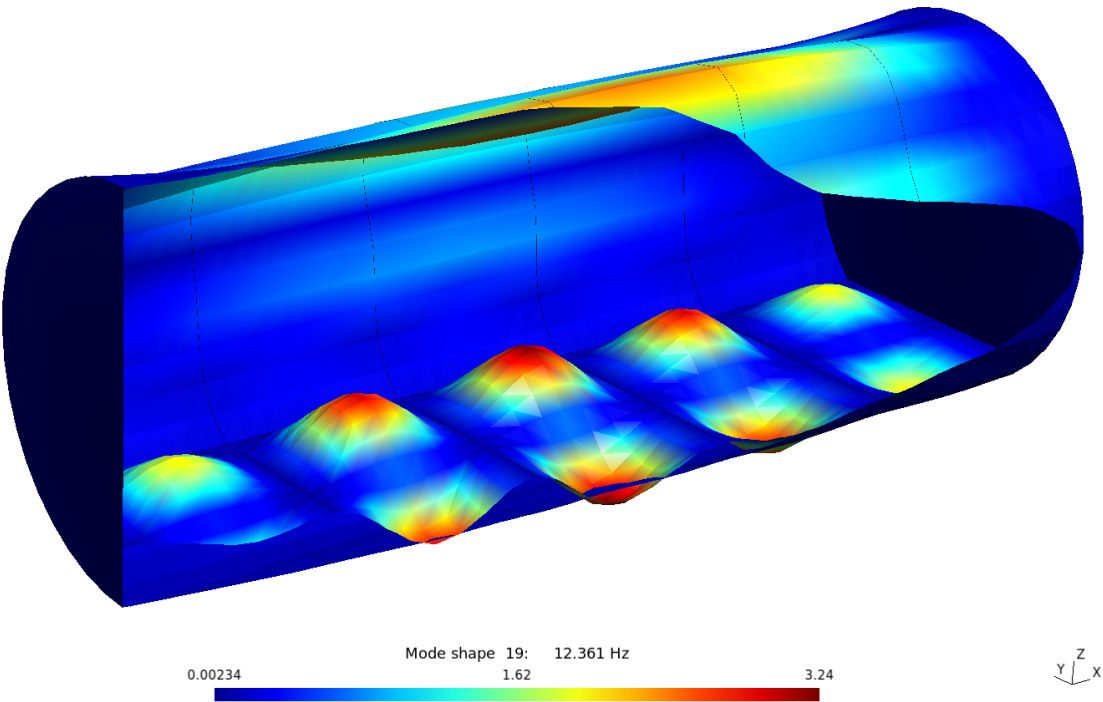


Figure 3.6-6: Cylindrical Box, Mode 19: 12.36 Hz

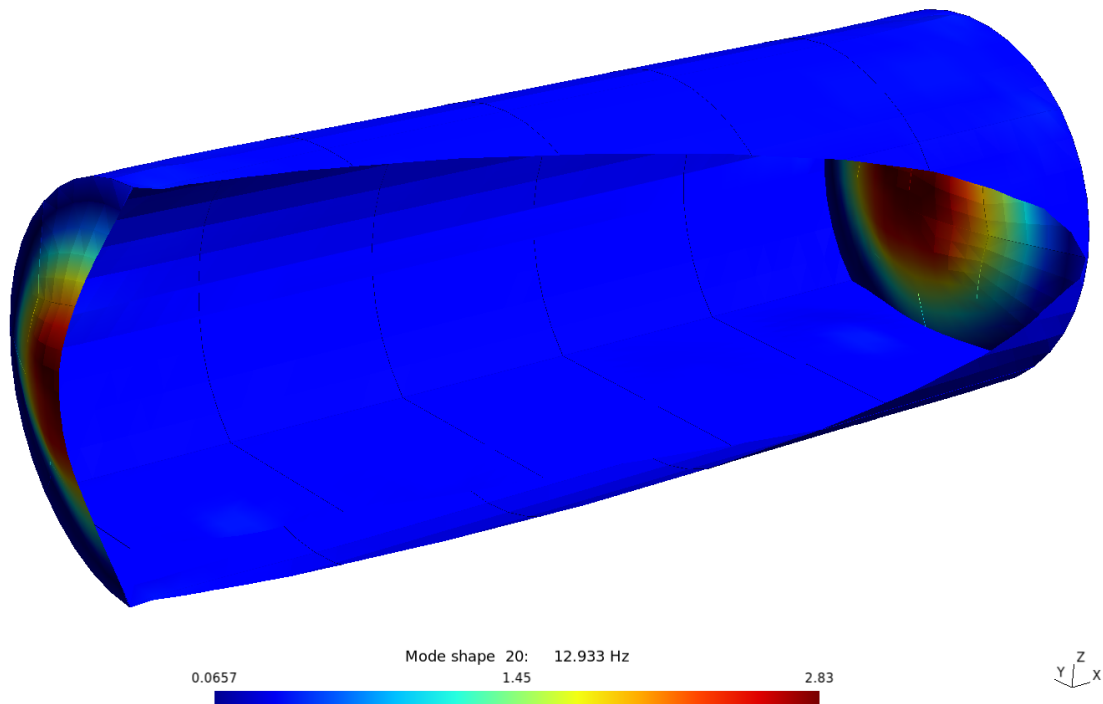


Figure 3.6-7: Cylindrical Box, Mode 20: 12.93 Hz

The figures show that the mesh is fine enough for all normal modes computed.



## 4 Frequency Response Analysis

### 4.1 2-dimensional Truss

#### Summary

Directory:	exa/solid/freqresp/truss2d
Objectives:	<ul style="list-style-type: none"> <li>• learn how to define force excitation</li> <li>• learn how to define modal damping ratios</li> <li>• learn how to assess the number of normal modes in a modal frequency response analysis</li> <li>• learn how to run a modal frequency response analysis</li> <li>• learn how to retrieve frequency response functions</li> </ul>
Dimension:	2
Elements:	<b>r2</b>
Loads:	concentrated nodal point forces
Functions:	<b>mfs_new, mfs_stiff, mfs_mass, mfs_massproperties, mfs_freevib, mfs_plot, mfs_print, mfs_reductionerror, mfs_freqresp, mfs_getresp</b>

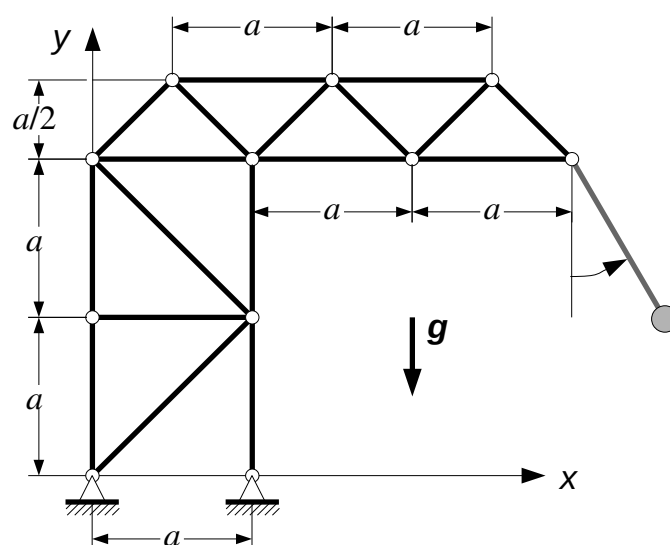


Figure 4.1-1: Truss with Swinging Load

### Problem Description

The truss structure already studied in Example 2.1 is loaded by a swinging load, see Figure 4.1-1. The swinging load leads to periodic forces in x- and y-direction acting on the structure. As periodic forces can be expanded into a Fourier series, the response can be computed if the response to harmonic forces is known.

To compute the response to harmonic forces, consider two load cases. Load case 1 has a unit load in x-direction and load case 2 in y-direction, both at node 8, see Figure 4.1-2.

First, compute the first 10 normal modes. Use the modal static strain energies to check if the number of modes is sufficient to compute the response up to a frequency of 400 Hz.

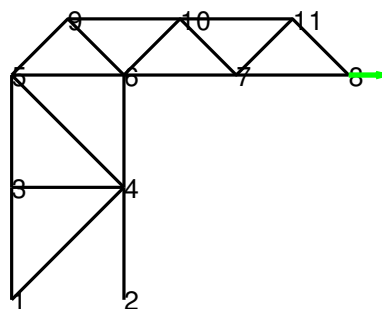
Next, perform an enhanced modal frequency response analysis to compute the modal responses of both load cases. Consider the frequency range from 10 Hz to 400 Hz.

Subsequently, get the displacements at the tip in x- and y-direction for both load cases and plot them as a function of the excitation frequency.

Finally, get the normal forces in elements 1 to 3 (see Figure 4.1-2) for both load cases and plot them as a function of the excitation frequency.

Data:  $a = 1000$  mm, cross section area  $A = 400$  mm<sup>2</sup>, Young's modulus

Loadcase 1:



Loadcase 2:

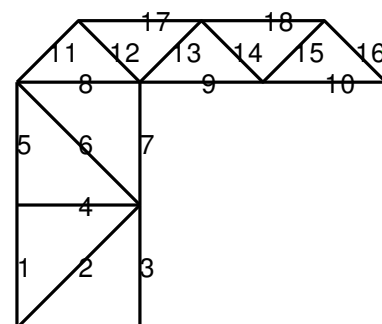


Figure 4.1-2: Truss, Finite Element Model with Loads

$E = 210$  GPa, modal damping ratios: 5 % for the first two normal modes and 2 % for all remaining normal modes

### Model Definition

File `truss.m` begins with the definition of the data needed:

```
# Data (N, mm)

F      =      1;    % Unit load
E      = 210000;    % Young's modulus in MPa
rho    = 7.85E-9;   % Mass density in t/mm^3
A      =      400;  % Cross section area in mm^2
a      =     1000;  % Length in mm
nofmod =      10;   % Number of normal modes
fmin   =      10;   % Lowest excitation frequency
fmax   =     400;   % Highest excitation frequency
df     =       1;   % Frequency increment

# Output file

fid = fopen("truss.res", "wt");
```

Subsequently, the model type, the nodes, the elements and the constraints are defined as in Example 2.1. However, the material data must include the mass density. As the units used are N and mm, the unit of the mass density is  $\text{t/mm}^3$ .

```
...
mat = struct("type", "iso", "E", E, "rho", rho);
...
```

Loads are defined as in a static analysis. In contrast to Example 2.1, we have to define two load cases.

```
# Loads

model.loads.point = struct("id",      8,
                           "data",    {[F, 0], [0, F]},
                           "lc",      {1, 2});
```

Finally, we define the damping. The damping type used is modal damping ratios. Modal damping ratios have to be assigned to normal modes. We can do this by using an array. The first two entries assign a value of 5 % to the first two normal modes. The third entry, which is the last, assigns a value of 2 % to all remaining normal modes.

```
# Damping

model.damping = struct("type", "ratios",
                       "data", [0.05, 0.05, 0.02]);
```

## Normal Modes Analysis

In a modal frequency response analysis, the normal modes have to be computed before the frequency response analysis can be performed.

First, the component is created and the model with the loads of the two load cases is plotted. In addition, the loads are written to the output file.

```
# Create and plot component

truss = mfs_new(fid, model);
h = figure(1, "position", [100, 100, 750, 500],
          "paperposition", [0, 0, 15, 10]);
subplot (1, 2, 1);
    mfs_plot(truss, "nodid", 1, "force", 1,
             "loadcase", 1, "figure", h);
subplot (1, 2, 2);
    mfs_plot(truss, "eltid", 1, "force", 1,
             "loadcase", 2, "figure", h);
print("model.svg", "-dsvg", "-F:10");
mfs_print(fid, truss, "loads", "point");
```

Next, the stiffness and the mass matrix are computed. In this example, we use a consistent mass matrix. Subsequently, the mass properties are computed and written to the output file.

```
# Compute normal modes

truss = mfs_stiff(truss);
truss = mfs_mass(truss, "consistent");
mfs_massproperties(fid, truss);
```

Then, the normal modes are computed and the resonance frequencies are written to the output file.

```
truss = mfs_freevib(truss, nofmod);
mfs_print(fid, truss, "modes", "freq");
```

Up to now, the output file contains the following data:

Mefisto 2.7: Building new component from input "model"

```
Model Type = solid, Model Subtype = 2d

Number of nodes      =    11,  Number of elements =    18
Number of element types =    1
Number of global      degrees of freedom =    22
Number of local      degrees of freedom =    18
Number of prescribed degrees of freedom =    4
Number of dependent  degrees of freedom =    0

Number of load cases =    2
```

-----

Component "truss"

Point loads of loadcase 1

node	Fx	Fy	
8	1.000e+00	0.000e+00	
Res.	1.000e+00	0.000e+00	-2.000e+03

Point loads of loadcase 2

node	Fx	Fy	
8	0.000e+00	1.000e+00	
Res.	0.000e+00	1.000e+00	3.000e+03

Mass properties of component "truss"

Coordinates of reference point: 0.0000, 0.0000

Rigid body mass matrix:

5.3603e-02	0.0000e+00	-8.9096e+01
0.0000e+00	5.3603e-02	5.5823e+01
-8.9096e+01	5.5823e+01	2.7111e+05

Mass = 5.3603e-02, Moment of inertia = 2.7111e+05

Coordinates of center of mass: 1041.4214, 1662.1320

Moment of inertia with respect to center of mass: 6.4890e+04

-----

Component "truss"

Natural frequencies:

Mode	Circ. Frequency	Frequency
1	346.47108	55.14258 Hz
2	535.29994	85.19563 Hz
3	1685.52119	268.25903 Hz
4	2084.04274	331.68570 Hz
5	3150.03744	501.34403 Hz
6	4122.68398	656.14553 Hz
7	4985.83903	793.52093 Hz
8	5668.41681	902.15655 Hz
9	6586.68606	1048.30365 Hz
10	6990.30747	1112.54199 Hz

Next, the modal static strain energies and an estimate of the error in strain energy at the highest excitation frequency are computed.

```
# Estimate reduction error
```

```
mfs_reductionerror(fid, truss, fmax);
```

The results of both load cases are written to the output file.

Modal strain energies of component "truss"

## Loadcase 1:

mode	frequency	En/ES	Sum	1 - Sum
1	55.14 Hz	4.55667e-01	0.455667	5.44333e-01
2	85.20 Hz	4.61802e-01	0.917469	8.25315e-02
3	268.26 Hz	3.36873e-04	0.917805	8.21946e-02
4	331.69 Hz	2.52560e-02	0.943061	5.69386e-02
5	501.34 Hz	1.80903e-03	0.944870	5.51295e-02
6	656.15 Hz	2.21078e-02	0.966978	3.30217e-02
7	793.52 Hz	1.71038e-03	0.968689	3.13114e-02
8	902.16 Hz	5.40043e-03	0.974089	2.59109e-02
9	1048.30 Hz	1.22578e-02	0.986347	1.36532e-02
10	1112.54 Hz	2.07675e-03	0.988424	1.15764e-02

Upper bound on relative strain energy error (fmax = 400.00 Hz)  
 with static correction: 1.8462e-03  
 without static correction: 7.6344e-03

## Loadcase 2:

mode	frequency	En/ES	Sum	1 - Sum
1	55.14 Hz	6.33811e-01	0.633811	3.66189e-01
2	85.20 Hz	3.29138e-01	0.962949	3.70507e-02
3	268.26 Hz	4.73539e-03	0.967685	3.23153e-02
4	331.69 Hz	1.63089e-02	0.983994	1.60064e-02
5	501.34 Hz	1.05927e-02	0.994586	5.41371e-03
6	656.15 Hz	8.36547e-05	0.994670	5.33006e-03
7	793.52 Hz	2.51767e-03	0.997188	2.81239e-03
8	902.16 Hz	2.10433e-05	0.997209	2.79135e-03
9	1048.30 Hz	1.43071e-03	0.998639	1.36064e-03
10	1112.54 Hz	7.02257e-07	0.998640	1.35994e-03

Upper bound on relative strain energy error (fmax = 400.00 Hz)  
 with static correction: 2.1688e-04  
 without static correction: 8.9685e-04

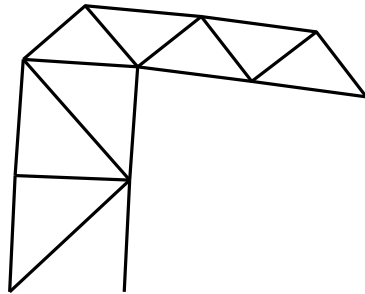
With static correction, the relative error in strain energy is less than 0.2 % for load case 1 and less than 0.022 % for load case 2. Thus, the number of normal modes is sufficient. It can be seen that for both load cases, the first two normal modes are the most important ones.

Finally, the first four normal modes are plotted in one plot. As this is a very small 2-dimensional model, we can use the Mefisto function `mfs_plot`. Figure 4.1-3 shows the resulting picture.

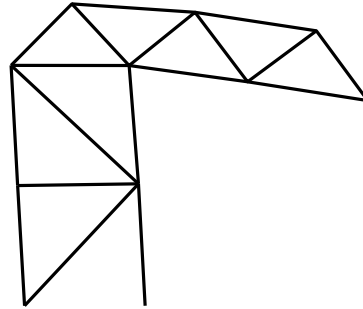
## Plot some modes

```
h = figure(2, "position", [200, 100, 500, 500],
           "paperposition", [0, 0, 15, 15]);
subplot(2, 2, 1);
mfs_plot(truss, "deform", 1, "modes", 1, "figure", h);
subplot(2, 2, 2);
mfs_plot(truss, "deform", 1, "modes", 2, "figure", h);
subplot(2, 2, 3);
mfs_plot(truss, "deform", 1, "modes", 3, "figure", h);
subplot(2, 2, 4);
mfs_plot(truss, "deform", 1, "modes", 4, "figure", h);
print("modes.svg", "-dsvg");
```

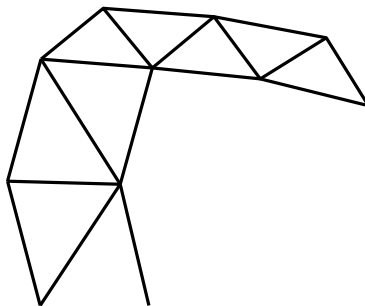
Mode 1: 55.143 Hz



Mode 2: 85.196 Hz



Mode 3: 268.259 Hz



Mode 4: 331.686 Hz

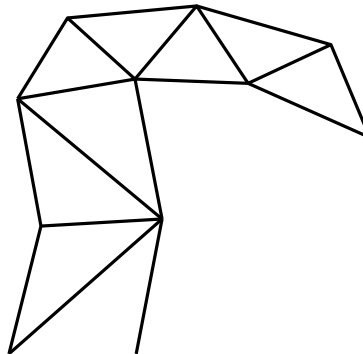


Figure 4.1-3: Truss, Normal Modes 1 to 4

### Frequency Response Analysis

File `truss.m` continues with the frequency response analysis. First, a set of equidistant excitation frequencies is defined in the frequency range from 10 Hz to 400 Hz. Then, function `mfs_freqresp` is called to compute the frequency response of the first load case. In addition to structure `truss` containing the component, the function returns a modified list of excitation frequencies, with 5 additional frequencies added within the half power bandwidth of each resonance frequency. The number of additional frequencies within the half power bandwidth can be controlled by parameter `nband`. The default of this parameter is 5, so it need not be defined here. Also, the default method is an enhanced modal frequency response analysis, with static approximation of the neglected modes, so no method need be defined.

Subsequently, function `mfs_freqresp` is called a second time to compute the frequency response of the second load case. Because the additional frequencies are already present from the output of the first call, they need not be determined again. Thus, parameter `nband` is given a value of 0.

```
# Compute frequency response of both load cases
```

```
f0 = fmin : df : fmax;
```

```
[truss, f] = mfs_freqresp(truss, f0, "loadcase", 1);  
truss      = mfs_freqresp(truss, f, "nband", 0, "loadcase", 2);
```

Now, function `mfs_getresp` can be used to retrieve the results of interest. First, we retrieve the modal coefficients of the first four normal modes and plot their frequency response.

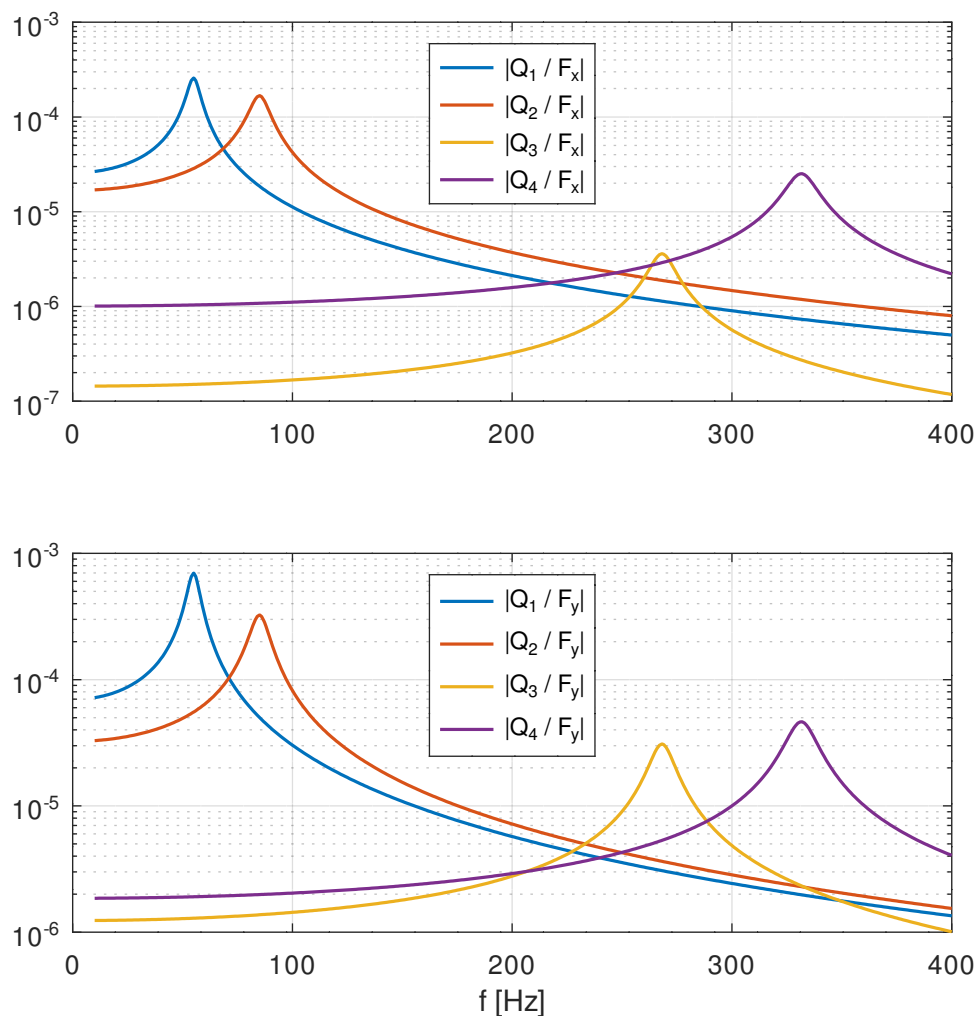


Figure 4.1-4: Truss, Frequency Response of Selected Normal Modes



```
# Get coefficients of first four normal modes

Q1 = mfs_getresp(truss, "freqresp", "Q", 1 : 4, 1);
Q2 = mfs_getresp(truss, "freqresp", "Q", 1 : 4, 2);

figure(3, "position", [300, 100, 750, 750],
       "paperposition", [0, 0, 15, 15]);
subplot(2, 1, 1);
semilogy(f, abs(Q1));
grid;
legend('|Q_1 / F_x|', '|Q_2 / F_x|', '|Q_3 / F_x|',
       '|Q_4 / F_x|',
       'location', 'north');
subplot(2, 1, 2);
semilogy(f, abs(Q2));
grid;
legend('|Q_1 / F_y|', '|Q_2 / F_y|', '|Q_3 / F_y|',
       '|Q_4 / F_y|',
       'location', 'north');
xlabel('f [Hz]');
print("Q.svg", "-dsvg");
```

The resulting plot can be seen in Figure 4.1-4. As expected, the response of each normal mode corresponds to the response of an SDOF system. It can be seen that the main contribution comes from the first two normal modes. These are the modes that also have the largest modal strain energies.

Next, we retrieve and plot the transfer functions of the displacements at the

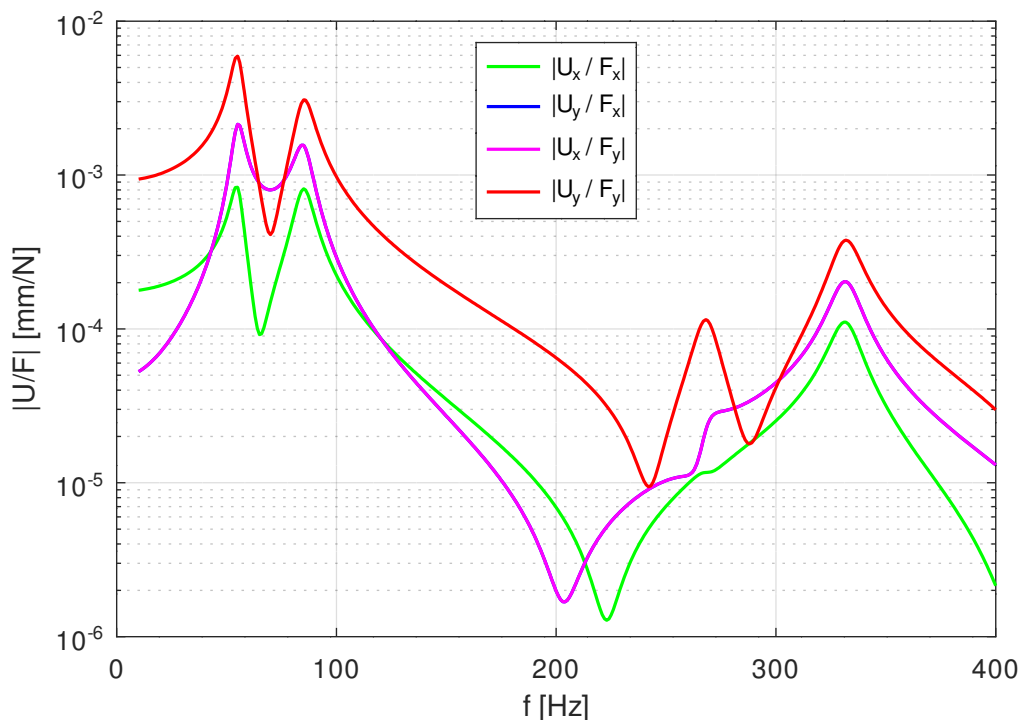


Figure 4.1-5: Truss, Transfer Functions of Displacements

tip where the loads are applied. Figure 4.1-2 shows that this is node 8. The resulting plot can be seen in Figure 4.1-5.

```
# Get transfer functions of tip displacements

rid = [8, 1; 8, 2];    % node id / dof

U1 = mfs_getresp(truss, "freqresp", "disp", rid, 1);
U2 = mfs_getresp(truss, "freqresp", "disp", rid, 2);

figure(4, "position", [400, 100, 750, 500],
        "paperposition", [0, 0, 15, 10]);
semilogy(f, abs(U1(1, :)), "color", "green",
         f, abs(U1(2, :)), "color", "blue",
         f, abs(U2(1, :)), "color", "magenta",
         f, abs(U2(2, :)), "color", "red");

grid;
legend('|U_x / F_x|', '|U_y / F_x|', '|U_x / F_y|',
       '|U_y / F_y|',
       'location', 'north')
xlabel('f [Hz]');
ylabel('|U/F| [mm/N]');
print("U.svg", "-dsvg");
```

Finally, we retrieve and plot the transfer functions of the normal forces in rods 1 to 3 (see Figure 4.1-2). For element results, function **mfs\_getresp** returns a cell array of structures, each structure containing the results of one element. The fields of the structures depend on the element type, as described in the User Manual. For rod elements, the only field with stress resultants is field **N** containing the normal forces. The resulting plot can be seen in Figure 4.1-6.

```
# Get transfer functions of normal forces in elements 1 to 3

N1 = mfs_getresp(truss, "freqresp", "resultant", 1 : 3, 1);
N2 = mfs_getresp(truss, "freqresp", "resultant", 1 : 3, 2);

figure(5, "position", [500, 100, 750, 750],
        "paperposition", [0, 0, 15, 15]);
subplot(2, 1, 1)
    semilogy(f, abs(N1{1}.N), "color", "green",
             f, abs(N1{2}.N), "color", "blue",
             f, abs(N1{3}.N), "color", "red");
    grid;
    legend('Rod 1', 'Rod 2', 'Rod 3',
           'location', 'southwest');
    ylabel('|N / F_x|');
subplot(2, 1, 2)
    semilogy(f, abs(N2{1}.N), "color", "green",
             f, abs(N2{2}.N), "color", "blue",
             f, abs(N2{3}.N), "color", "red");
    grid;
    xlabel('f [Hz]');
```

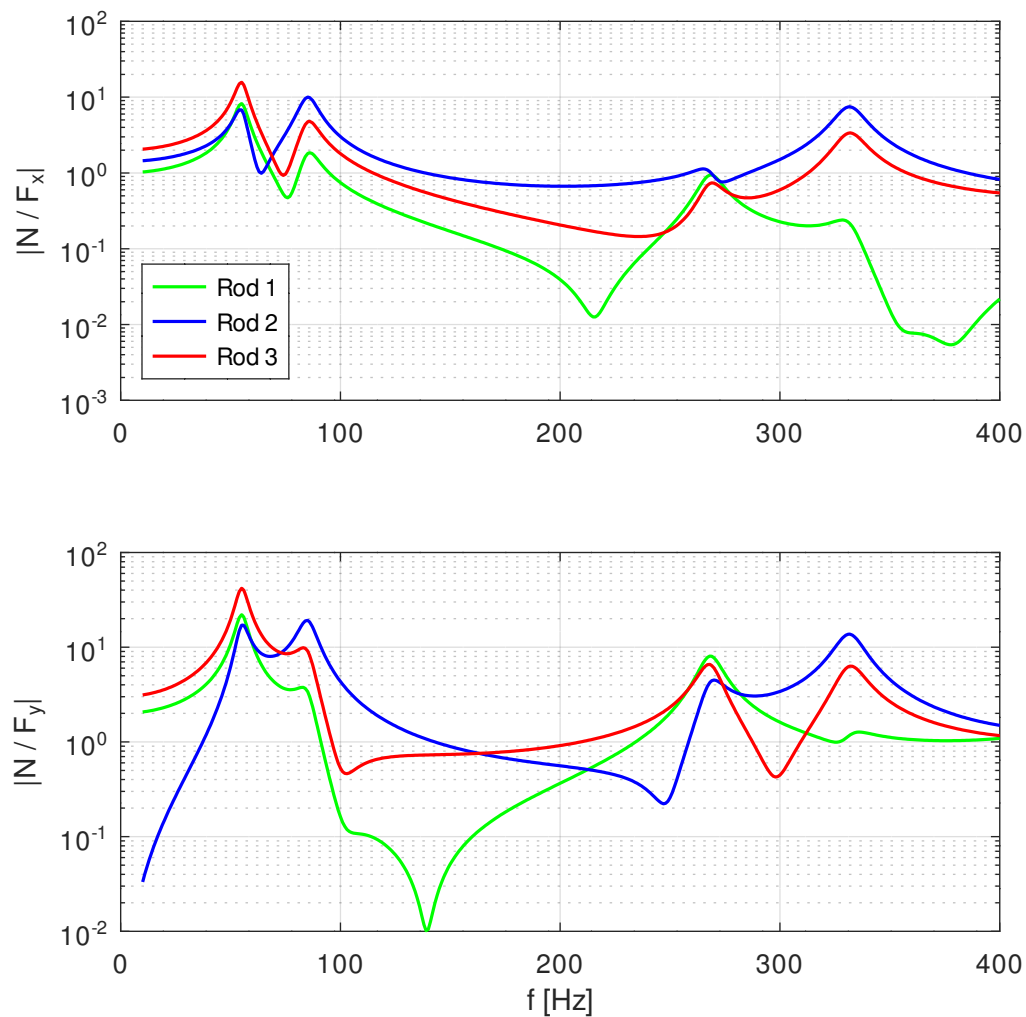


Figure 4.1-6: Truss, Transfer Functions of Normal Forces

```

ylabel(' |N / F_y| ');
print("N.svg", "-dsvg");

fclose(fid);

```

## 4.2 Box Beam

### Summary

Directory:	exa/solid/freqresp/boxbeam
Objectives:	<ul style="list-style-type: none"> <li>• learn how to define force excitation</li> <li>• learn how to define prescribed motion</li> </ul>

	<ul style="list-style-type: none"> <li>• learn how to define Rayleigh damping</li> <li>• learn how to assess the number of normal modes in a modal frequency response analysis</li> <li>• learn how to work with node and element sets</li> <li>• learn how to run a modal frequency response analysis</li> <li>• learn how to run a direct frequency response analysis</li> <li>• learn how to retrieve frequency response functions</li> <li>• learn how to back-transform results from a modal frequency response analysis</li> <li>• learn how to work with restarts</li> <li>• learn how to compute and interpret modal contribution factors</li> </ul>
Dimension:	3
Elements:	<b>s4</b>
Loads:	<ul style="list-style-type: none"> <li>• concentrated nodal point forces</li> <li>• prescribed acceleration</li> </ul>
Functions:	<code>mfs_import, mfs_new, mfs_stiff, mfs_mass, mfs_massproperties, mfs_freevib, mfs_print, mfs_export, mfs_reductionerror, mfs_meffmass, mfs_newset, mfs_freqresp, mfs_getresp, mfs_peaks, mfs_back, mfs_results, mfs_modecont</code>

### Problem Description

The cantilever beam with a thin-walled box cross section is subjected to the following two excitations (cf. Figure 4.2-1):

1. a vertical force of 1 N at point A
2. a vertical acceleration of 1 g of the restrained left end

The frequency range to be considered is from 20 Hz to 800 Hz.

For both load cases, compute

1. the vertical displacements at points A and B
2. the normal stresses in x-direction at the upper side of the upper shell at points C and D

In a first step, based on the strain energy error and the modal effective masses, determine the number of normal modes needed for the modal reduction. Use both a modal and an enhanced modal reduction and compare the results with those from a direct frequency response analysis.

Data:  $a = 1000$  mm,  $b = 100$  mm,  $h = 60$  mm,  $t = 3$  mm, Young's modulus  $E =$

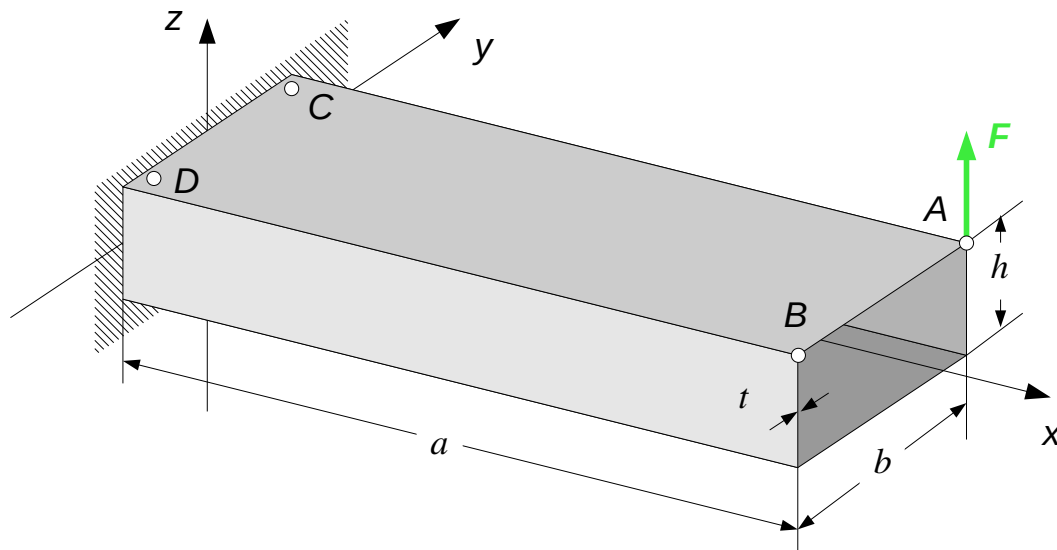


Figure 4.2-1: Box Beam Structure

210 GPa, Poisson's ratio  $\nu = 0,3$ , mass density  $\rho = 7850 \text{ kg/m}^3$ , Rayleigh damping with  $\alpha_K = 10^{-5} \text{ s}$ ,  $\alpha_M = 20 \text{ s}^{-1}$

### Model Definition

The model is defined in Gmsh. The following physical groups are defined:

- **Shell**: shell elements
- **Constraints**: used to apply the constraints at the left end
- **Load**: used to apply the force at point A
- **Motion**: used to apply the prescribed acceleration at the left end
- **A, B**: used to create node sets corresponding to points A and B

The commands to define the geometry and the mesh are contained in file beam.geo:

```
/* -----
                        Box Beam Structure
----- */

// Dimensions [mm]

DefineConstant [ a = 1000, // Length
                  b = 100,  // Width
                  h = 60]; // Height

// Discretization
```

```
DefineConstant [ na = 50,    // Number of elem. in x-direction
                 nb = 7,     // Number of elem. in y-direction
                 nc = 5];    // Number of elem. in z-direction

Mesh.ElementOrder = 1;
Mesh.RecombineAll = 1;
Mesh.Smoothing     = 1;

// Geometry

y0 = 0.5 * b;
z0 = 0.5 * h;

Point(1) = {0, -y0, -z0};
Point(2) = {a, -y0, -z0};
Point(3) = {a,  y0, -z0};
Point(4) = {0,  y0, -z0};

Point(5) = {0, -y0,  z0};
Point(6) = {a, -y0,  z0};
Point(7) = {a,  y0,  z0};
Point(8) = {0,  y0,  z0};

Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 1};

Line(5) = {5, 6};
Line(6) = {6, 7};
Line(7) = {7, 8};
Line(8) = {8, 5};

Line(9) = {1, 5};
Line(10) = {4, 8};
Line(11) = {2, 6};
Line(12) = {3, 7};

Curve Loop(1) = {1, 2, 3, 4};
Plane Surface(1) = {1};
Curve Loop(2) = {5, 6, 7, 8};
Plane Surface(2) = {2};
Curve Loop(3) = {1, 11, -5, -9};
Plane Surface(3) = {3};
Curve Loop(4) = {3, 10, -7, -12};
Plane Surface(4) = {4};

// Shell Elements

Physical Surface("Shell") = {1 : 4};

// Constraints

Physical Curve("Constraints") = {4, 8, 9, 10};
```

```
// Load
Physical Point("Load") = {7};

// Base motion
Physical Curve("Motion") = {4, 8, 9, 10};

// Response
Physical Point("A") = {7};
Physical Point("B") = {6};

// Structured Mesh
Transfinite Line{1, 3, 5, 7} = na + 1;
Transfinite Line{2, 4, 6, 8} = nb + 1;
Transfinite Line{9 : 12}      = nc + 1;

Transfinite Surface{1 : 4};
```

Figure 4.2-2 shows the resulting finite element mesh.

File `modes.m` contains the model definition and the commands to perform the normal modes analysis. First, the output file is opened and the data needed are defined.

```
fid = fopen("modes.res", "wt");

# Data (N, mm):

E   = 210000; % Young's modulus
ny  = 0.3;    % Poisson's ratio
rho = 7.85E-9; % Mass density
t   = 3;      % Thickness
```

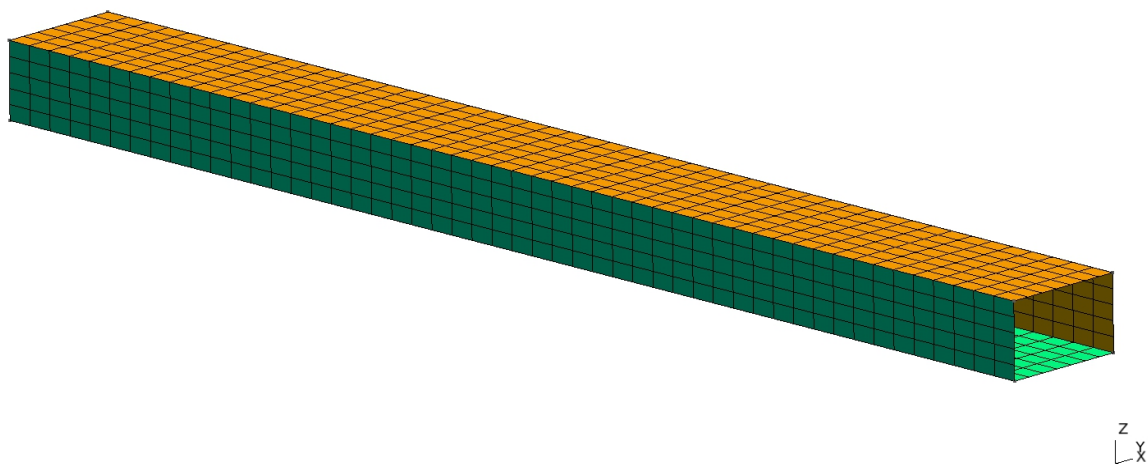


Figure 4.2-2: Box Beam, Finite Element Mesh

```

F    =    1;    % Load
az   =   9810;  % Prescribed vertical acceleration (1 g)

aK   =   1e-5;  % Parameter for Rayleigh-damping
aM   =    20;  % Parameter for Rayleigh-damping

nofmod = 20;    % Number of normal modes
fmax   = 800;   % Maximum excitation frequency

# Translation data

mat   = struct("type", "iso", "E", E, "ny", ny, "rho", rho);
geom  = struct("t", t);

BEAM = struct("type", "solid", "subtype", "3d");

```

Next, the translation data for the elements, constraints, loads and the prescribed motion are defined. Force excitation is load case 1 and base motion excitation is load case 2.

```

BEAM.Shell = struct("type", "elements", "name", "s4",
                    "geom", geom, "mat", mat);

BEAM.Constraints = struct("type", "constraints",
                          "name", "prescribed",
                          "dofs", 1 : 6);

BEAM.Load = struct("type", "loads",
                   "name", "point",
                   "data", [0, 0, F],
                   "lc", 1);

BEAM.Motion = struct("type", "loads",
                    "name", "acce",
                    "data", [0, 0, az],
                    "lc", 2);

```

The damping is Rayleigh damping, i.e. the damping matrix reads

$$[D] = \alpha_K [K] + \alpha_M [M].$$

In contrast to modal damping ratios, Rayleigh damping can also be used in a direct frequency response analysis. Its definition reads:

```

BEAM.damping = struct("type", "Rayleigh",
                      "data", [aK, aM]);

```

Finally, two node sets are defined that contain the nodal point identifiers corresponding to points *A* and *B* where we want to obtain the displacements.

```

BEAM.A.type = "nodeset";
BEAM.B.type = "nodeset";

```



## Normal Modes Analysis

Before computing the frequency response, we have to calculate the normal modes and check the reduction error. From the normal modes analysis, we will know the resonance frequencies, and we can check if the discretization is fine enough.

File `modes.m` continues with the commands to perform a normal modes analysis. First, the mesh definition is imported from file `beam.msh`.

```
# Import model

model = mfs_import(fid, "beam.msh", "msh", BEAM);
```

Next, the component is created and the element axes are exported so that we can display them in Gmsh. The normal vectors of the elements of the upper surface of the box point in positive z-direction, so the stresses we are interested in are those of the upper surface of the elements.

```
# Create component and export axes

beam = mfs_new(fid, model);
mfs_export("beam.axes", "msh", beam, "mesh", "axes");
```

Then the stiffness and mass matrices are computed. In addition, the mass properties are computed and written to the output file.

```
# Compute normal modes

beam = mfs_stiff(beam);
beam = mfs_mass(beam);
mfs_massproperties(fid, beam);
```

Subsequently, the normal modes are computed. The resonance frequencies are written to the output file, and the mode shapes are exported to Gmsh.

```
beam = mfs_freevib(beam, nofmod);
mfs_print(fid, beam, "modes", "freq");
mfs_export("modes.dsp", "msh", beam, "modes", "disp");
```

Function `mfs_reductionerror` computes the modal static strain energies and estimates the error in strain energy at the highest excitation frequency `fmax`. The results are computed for both load cases and written to the output file.

```
# Estimate reduction error
```

```
mfs_reductionerror(fid, beam, fmax);
```

Then the modal effective masses are computed and written to the output file. Modal effective masses can be used to evaluate the modal basis if the excitation is a rigid body base motion excitation.

```
# Compute modal effective masses
```

```
mfs_meffmass(fid, beam);
```

To access the normal stresses we define two element sets. As it is complicated to define these sets based on physical properties, we use function **mfs\_newset** to define them based on the geometry.

```
# Elements for stress output
```

```
beam = mfs_newset(beam, "eset", "near", [5, 40, 30], "C");
beam = mfs_newset(beam, "eset", "near", [5, -40, 30], "D");
```

Set **C** contains the element whose midpoint is closest to the point with coordinates **[5, 40, 30]** and set **D** the element whose midpoint is closest to the point with coordinates **[5, -40, 30]**.

Finally, the component together with the highest excitation frequency is saved, and the output file is closed.

```
# Save results
```

```
save -binary modes.bin fmax beam
```

```
fclose(fid);
```

The output file **modes.res** contains the resonance frequencies, the mass properties, the modal strain energies and the modal effective masses.

Reading model from file "beam.msh", MSH file version 4.1

Physical Group	Type
Shell	elements
Constraints	constraints
Load	loads
Motion	loads
A	nodeset
B	nodeset

Mefisto 2.7: Building new component from input "model"

```
Model Type = solid, Model Subtype = 3d
```

```
Number of nodes      = 1224,  Number of elements = 1200
Number of element types = 1
Number of global      degrees of freedom = 7344
Number of local       degrees of freedom = 7200
Number of prescribed degrees of freedom = 144
```

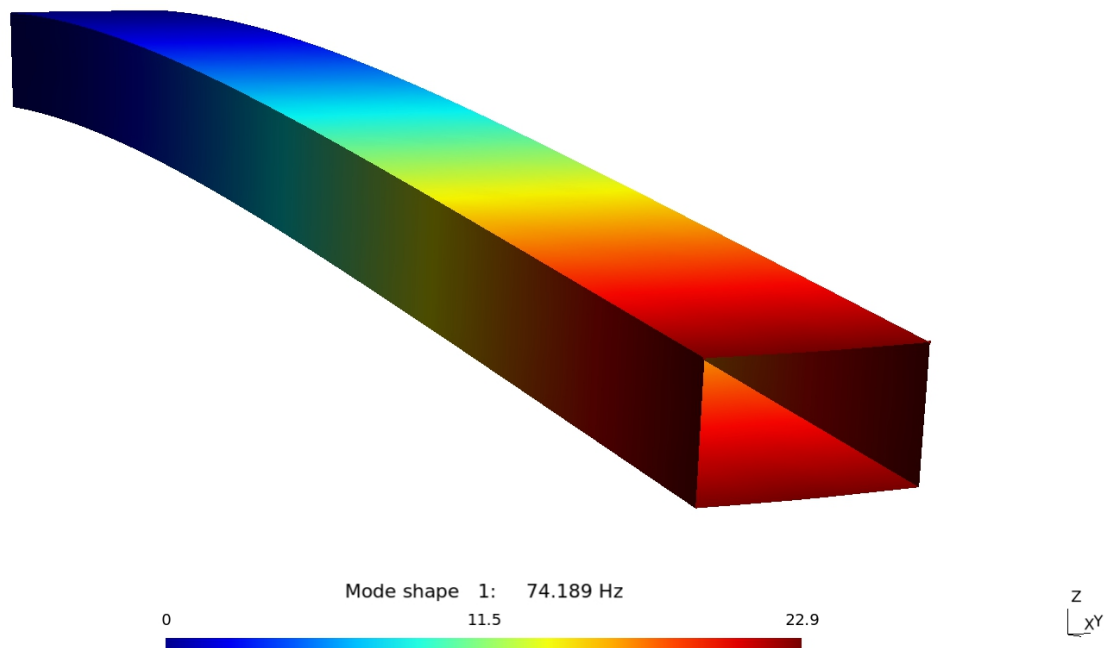


Figure 4.2-3: Box Beam, 1. Vertical Bending Mode

Number of dependent degrees of freedom = 0

Number of load cases = 2

Mass properties of component "beam"

Coordinates of reference point: 0.0000, 0.0000, 0.0000

Rigid body mass matrix:

7.5360e-03	0.0000e+00	0.0000e+00	0.0000e+00	-6.8056e-18	6.9389e-18
0.0000e+00	7.5360e-03	0.0000e+00	6.8056e-18	0.0000e+00	3.7680e+00
0.0000e+00	0.0000e+00	7.5360e-03	-1.0408e-17	-3.7680e+00	0.0000e+00
0.0000e+00	6.9118e-18	-2.0817e-17	1.6305e+01	-2.4269e-13	-4.5073e-15
-6.9118e-18	0.0000e+00	-3.7680e+00	-2.4469e-13	2.5177e+03	-3.3583e-17
1.0408e-17	3.7680e+00	0.0000e+00	-4.4501e-15	-4.7870e-18	2.5237e+03

Mass = 7.5360e-03

Inertia tensor with respect to reference point:

1.6305e+01	-2.4269e-13	-4.5073e-15
-2.4469e-13	2.5177e+03	-3.3583e-17
-4.4501e-15	-4.7870e-18	2.5237e+03

Coordinates of center of mass: 500.0000, -0.0000, -0.0000

Inertia tensor with respect to center of mass:

1.6305e+01	-2.4790e-13	-7.9101e-15
-2.4990e-13	6.3366e+02	-3.3583e-17
-7.8529e-15	-4.7870e-18	6.3965e+02

Component "beam"

Natural frequencies:

Mode	Circ. Frequency	Frequency
-----		

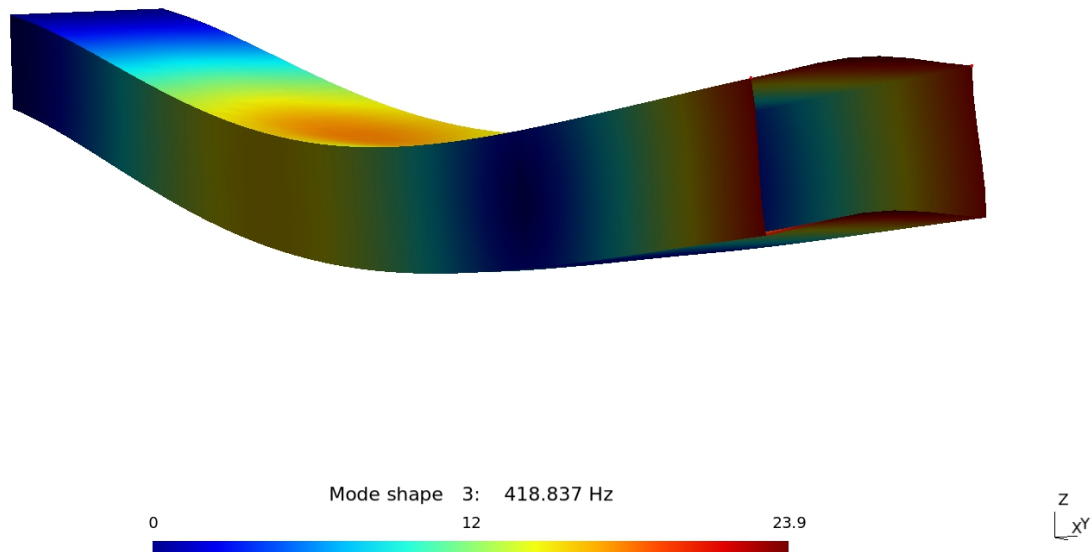


Figure 4.2-4: Box Beam, 2. Vertical Bending Mode

1	466.14244	74.18887 Hz
2	683.15069	108.72681 Hz
3	2631.62979	418.83689 Hz
4	3494.30891	556.13654 Hz
5	3869.09488	615.78558 Hz
6	4513.55247	718.35419 Hz
7	5264.42299	837.85894 Hz
8	5794.44498	922.21456 Hz
9	6390.22607	1017.03607 Hz
10	6471.17932	1029.92018 Hz
11	6621.83071	1053.89709 Hz
12	6654.64878	1059.12025 Hz
13	6849.63355	1090.15304 Hz
14	7156.54925	1139.00019 Hz
15	7544.84853	1200.79994 Hz
16	7598.90304	1209.40298 Hz
17	8017.00153	1275.94542 Hz
18	8133.31028	1294.45654 Hz
19	8371.53432	1332.37107 Hz
20	8575.64469	1364.85624 Hz

Modal strain energies of component "beam"

Loadcase 1:

mode	frequency	En/ES	Sum	1 - Sum
1	74.19 Hz	9.28045e-01	0.928045	7.19549e-02
2	108.73 Hz	4.14426e-11	0.928045	7.19549e-02
3	418.84 Hz	2.42195e-02	0.952265	4.77354e-02
4	556.14 Hz	1.61656e-02	0.968430	3.15698e-02
5	615.79 Hz	1.10664e-09	0.968430	3.15698e-02
6	718.35 Hz	1.30485e-02	0.981479	1.85213e-02
7	837.86 Hz	1.92238e-04	0.981671	1.83290e-02
8	922.21 Hz	2.39120e-03	0.984062	1.59378e-02
9	1017.04 Hz	4.22252e-08	0.984062	1.59378e-02
10	1029.92 Hz	2.18235e-08	0.984062	1.59378e-02
11	1053.90 Hz	2.68879e-08	0.984062	1.59377e-02
12	1059.12 Hz	4.14536e-03	0.988208	1.17924e-02
13	1090.15 Hz	3.35654e-08	0.988208	1.17923e-02

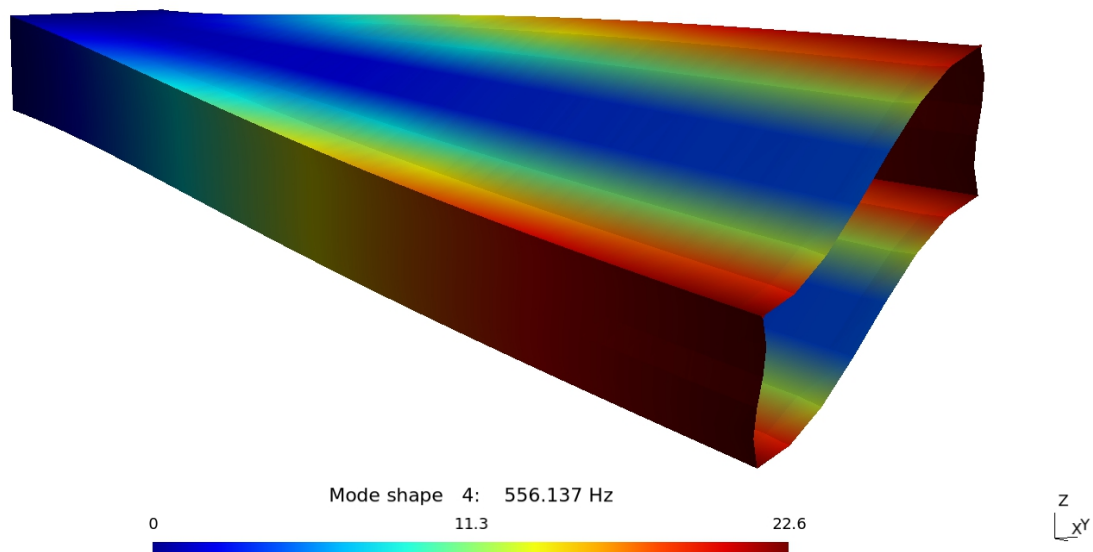


Figure 4.2-5: Box Beam, 1. Vertical Shear Mode

14	1139.00 Hz	4.24334e-08	0.988208	1.17923e-02
15	1200.80 Hz	5.33403e-08	0.988208	1.17923e-02
16	1209.40 Hz	3.13292e-04	0.988521	1.14790e-02
17	1275.95 Hz	6.58270e-08	0.988521	1.14789e-02
18	1294.46 Hz	4.09126e-14	0.988521	1.14789e-02
19	1332.37 Hz	6.84397e-05	0.988590	1.14105e-02
20	1364.86 Hz	7.89993e-08	0.988590	1.14104e-02

Upper bound on relative strain energy error (fmax = 800.00 Hz)  
 with static correction: 7.5347e-03  
 without static correction: 1.3240e-02

Loadcase 2:

mode	frequency	En/ES	Sum	1 - Sum
1	74.19 Hz	9.89058e-01	0.989058	1.09421e-02
2	108.73 Hz	3.92198e-25	0.989058	1.09421e-02
3	418.84 Hz	9.98749e-03	0.999045	9.54587e-04
4	556.14 Hz	1.89146e-30	0.999045	9.54587e-04
5	615.79 Hz	2.36914e-31	0.999045	9.54587e-04
6	718.35 Hz	1.45051e-30	0.999045	9.54587e-04
7	837.86 Hz	2.33152e-33	0.999045	9.54587e-04
8	922.21 Hz	6.63965e-04	0.999709	2.90622e-04
9	1017.04 Hz	3.48561e-29	0.999709	2.90622e-04
10	1029.92 Hz	2.26824e-29	0.999709	2.90622e-04
11	1053.90 Hz	1.78378e-31	0.999709	2.90622e-04
12	1059.12 Hz	6.29158e-31	0.999709	2.90622e-04
13	1090.15 Hz	2.73815e-30	0.999709	2.90622e-04
14	1139.00 Hz	1.13021e-29	0.999709	2.90622e-04
15	1200.80 Hz	9.46299e-30	0.999709	2.90622e-04
16	1209.40 Hz	1.44108e-04	0.999853	1.46514e-04
17	1275.95 Hz	7.98961e-30	0.999853	1.46514e-04
18	1294.46 Hz	1.99404e-32	0.999853	1.46514e-04
19	1332.37 Hz	4.92455e-05	0.999903	9.72680e-05
20	1364.86 Hz	3.48473e-31	0.999903	9.72680e-05

Upper bound on relative strain energy error (fmax = 800.00 Hz)  
 with static correction: 6.4230e-05  
 without static correction: 1.1286e-04

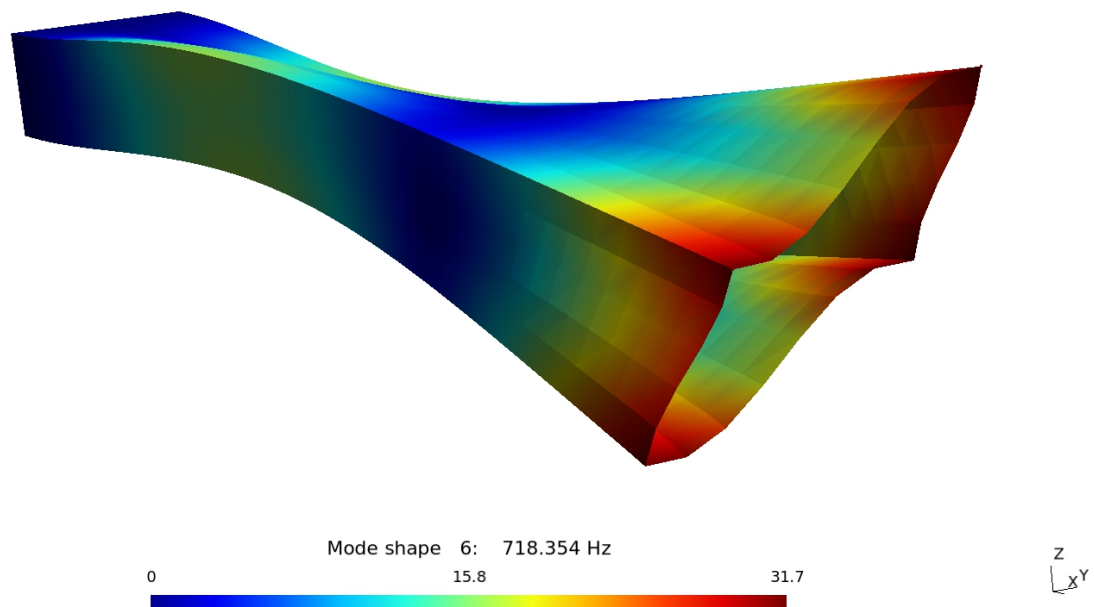


Figure 4.2-6: Box Beam, 2. Vertical Shear Mode

The modal static strain energies show that the number of normal modes is sufficient, although the resonance frequency of the highest mode computed is only about 70 % larger than the highest excitation frequency. With static correction, the relative strain energy error is less than 0,76 % for the first load case and less than 0,007 % for the second load case. Static approximation of the neglected modes is the default method of `mfs_freqresp`.

The most important normal modes for load case 1 can be seen to be modes 1, 3, 4, and 6. These modes are shown in Figures 4.2-3 to 4.2-6. For load case 2, the most important normal modes are modes 1 and 3.

-----

Modal effective masses of component "beam"

Coordinates of reference point: 0.0000, 0.0000, 0.0000

mode	frequency	tx	ty	tz	sum tx	sum ty	sum tz
1	74.19 Hz	1.3747e-29	2.2801e-26	6.2331e-01	0.00000	0.00000	0.62331
2	108.73 Hz	2.0067e-29	6.2139e-01	2.4444e-26	0.00000	0.62139	0.62331
3	418.84 Hz	8.6482e-30	8.4560e-30	2.0061e-01	0.00000	0.62139	0.82392
4	556.14 Hz	1.5933e-28	8.7947e-28	4.3691e-27	0.00000	0.62139	0.82392
5	615.79 Hz	4.5267e-30	2.0149e-01	6.4104e-29	0.00000	0.82288	0.82392
6	718.35 Hz	1.8726e-29	7.2010e-27	1.7794e-28	0.00000	0.82288	0.82392
7	837.86 Hz	1.0031e-31	1.3789e-27	9.7532e-30	0.00000	0.82288	0.82392
8	922.21 Hz	7.0415e-30	2.5996e-30	6.4657e-02	0.00000	0.82288	0.88858
9	1017.04 Hz	1.3310e-05	9.2466e-29	3.9006e-27	0.00001	0.82288	0.88858
10	1029.92 Hz	3.7045e-05	3.3553e-28	2.0739e-27	0.00005	0.82288	0.88858
11	1053.90 Hz	5.3338e-06	1.9458e-28	9.6802e-30	0.00006	0.82288	0.88858
12	1059.12 Hz	1.5750e-29	1.2578e-29	9.3720e-28	0.00006	0.82288	0.88858
13	1090.15 Hz	6.8457e-06	1.1959e-29	6.5350e-28	0.00006	0.82288	0.88858
14	1139.00 Hz	4.7442e-06	2.2545e-28	2.1808e-27	0.00007	0.82288	0.88858
15	1200.80 Hz	1.3689e-05	7.9426e-33	1.4320e-27	0.00008	0.82288	0.88858
16	1209.40 Hz	5.8440e-30	7.3326e-30	2.4134e-02	0.00008	0.82288	0.91271
17	1275.95 Hz	1.9292e-04	9.9206e-29	1.2961e-27	0.00027	0.82288	0.91271

18	1294.46 Hz	8.1689e-01	1.7248e-29	2.1422e-29	0.81716	0.82288	0.91271
19	1332.37 Hz	3.0170e-29	6.5693e-30	1.0010e-02	0.81716	0.82288	0.92272
20	1364.86 Hz	1.4579e-05	1.5614e-29	1.9374e-28	0.81718	0.82288	0.92272
mode	frequency	rx	ry	rz	sum rx	sum ry	sum rz
1	74.19 Hz	3.3708e-27	9.7263e-01	3.7276e-26	0.00000	0.97263	0.00000
2	108.73 Hz	1.3795e-27	3.8034e-26	9.7233e-01	0.00000	0.97263	0.97233
3	418.84 Hz	1.4362e-27	2.3768e-02	5.1626e-30	0.00000	0.99640	0.97233
4	556.14 Hz	5.6039e-01	6.4147e-27	1.7164e-28	0.56039	0.99640	0.97233
5	615.79 Hz	1.2125e-26	1.1310e-28	2.3997e-02	0.56039	0.99640	0.99632
6	718.35 Hz	4.3212e-02	1.6667e-29	9.8570e-29	0.60360	0.99640	0.99632
7	837.86 Hz	2.2206e-01	2.8820e-30	3.6467e-27	0.82567	0.99640	0.99632
8	922.21 Hz	1.5631e-27	2.4458e-03	7.8125e-31	0.82567	0.99885	0.99632
9	1017.04 Hz	3.1113e-28	4.1018e-29	5.7381e-30	0.82567	0.99885	0.99632
10	1029.92 Hz	1.0215e-28	4.5333e-29	1.7831e-28	0.82567	0.99885	0.99632
11	1053.90 Hz	1.7952e-26	8.3478e-28	1.6837e-28	0.82567	0.99885	0.99632
12	1059.12 Hz	8.3250e-03	1.2647e-27	2.6073e-28	0.83399	0.99885	0.99632
13	1090.15 Hz	6.2202e-28	1.3030e-27	2.0951e-29	0.83399	0.99885	0.99632
14	1139.00 Hz	1.1211e-28	7.7705e-30	4.5698e-29	0.83399	0.99885	0.99632
15	1200.80 Hz	5.9408e-28	1.1225e-28	1.7394e-30	0.83399	0.99885	0.99632
16	1209.40 Hz	2.3099e-32	3.6731e-04	9.3528e-31	0.83399	0.99922	0.99632
17	1275.95 Hz	4.8319e-28	1.7097e-28	5.4211e-29	0.83399	0.99922	0.99632
18	1294.46 Hz	6.8399e-29	2.6995e-29	2.2791e-29	0.83399	0.99922	0.99632
19	1332.37 Hz	1.0088e-27	7.2712e-05	2.5869e-31	0.83399	0.99929	0.99632
20	1364.86 Hz	8.7739e-31	1.1874e-28	9.0417e-31	0.83399	0.99929	0.99632

The modal effective masses confirm that modes 1 and 3 are the most important modes for load case 2, a vertical acceleration of the clamped end. These modes have the largest contribution to the modal effective mass in z-direction and the modal effective moment of inertia about the y-axis.

### Frequency Response Analysis

File `freqresp.m` contains the commands to perform an enhanced modal frequency response analysis, that is a modal frequency response analysis with a static approximation of the neglected modes.

First, some graphics parameters are defined and the component with the results of the normal modes analysis is loaded.

```
colors = [0, 1, 0; 1, 0, 0; 0, 0, 1]; % green; red; blue
set(0, "defaultaxescolororder", colors);

# Results from normal modes analysis

load modes.bin
```

Next, we define the excitation frequencies and perform the frequency response analysis. Remember that the maximum excitation frequency `fmax` has been stored in file `modes.bin`.

Function `mfs_freqresp` has to be called for each load case. In the first call, in addition to component `beam` with the results of the frequency response analysis of the first load case added, also an array `f` is returned which contains the excitation frequencies with additional excitation frequencies in the



halfpower bandwidth of each resonance frequency added. This list of excitation frequencies is used as input in the second call to compute the results of the second load case, so this time no more additional excitation frequencies need be added.

Please note that no method is defined. Thus, the default method is used, i.e. the enhanced modal frequency response analysis.

```
# Excitation frequencies

fmin = 20; % Minimum excitation frequency
df    = 5; % Frequency step
nb    = 7; % Additional frequencies in halfpower bandwidth

fe    = fmin : df : fmax; % fmax from modes.bin

# Frequency response analysis

[beam, f] = mfs_freqresp(beam, fe, "nband", nb, "loadcase", 1);
beam      = mfs_freqresp(beam, f, "nband", 0, "loadcase", 2);
```

After the frequency response analysis has been performed, we can use function **mfs\_getresp** to retrieve the transfer functions of interest. First, we look at the transfer functions for the vertical displacements at points A and B. We can use sets A and B to define the response identifiers. Cell array **rid** contains pairs of set names and degree of freedom identifiers. The last input argument of function **mfs\_getresp** is the load case identifier.

The nodal point identifiers are included in the legend so that we can check if they are correct. To obtain these identifiers we use function **mfs\_getset**.

```
# Transfer functions for displacements

rid = {"A", 3; "B", 3};

Uz1 = mfs_getresp(beam, "freqresp", "disp", rid, 1);
Uz2 = mfs_getresp(beam, "freqresp", "disp", rid, 2);

idA = mfs_getset(beam, "nset", "A");
idB = mfs_getset(beam, "nset", "B");
lgtext = {sprintf("Point A (%d)", idA), ...
          sprintf("Point B (%d)", idB)};

figure(1, "position", [100, 200, 1000, 750],
       "paperposition", [0, 0, 15, 15]);
subplot(2, 1, 1)
    semilogy(f, abs(Uz1));
    title("Loadcase 1");
    legend(lgtext);
    grid;
    ylabel('|U_z| / F [mm/N]');
subplot(2, 1, 2)
    semilogy(f, abs(Uz2));
```



```

title("Loadcase 2");
grid;
xlabel('f [Hz]');
ylabel('|U_z| / a_z [mm/g]');
print("U.svg", "-dsvg", "-F:10");

```

Figure 4.2-7 shows the transfer functions of the vertical displacements for both load cases. As the excitation of the second load case, the base motion excitation, is symmetric, the transfer functions for points A and B are identical. We can also observe that the peaks occur at the resonance frequencies of those normal modes that, based on the modal strain energies, we identified as the most important ones.

Next, we retrieve and plot the transfer functions of the stresses. To access the elements of interest we use sets C and D. The names of these sets are

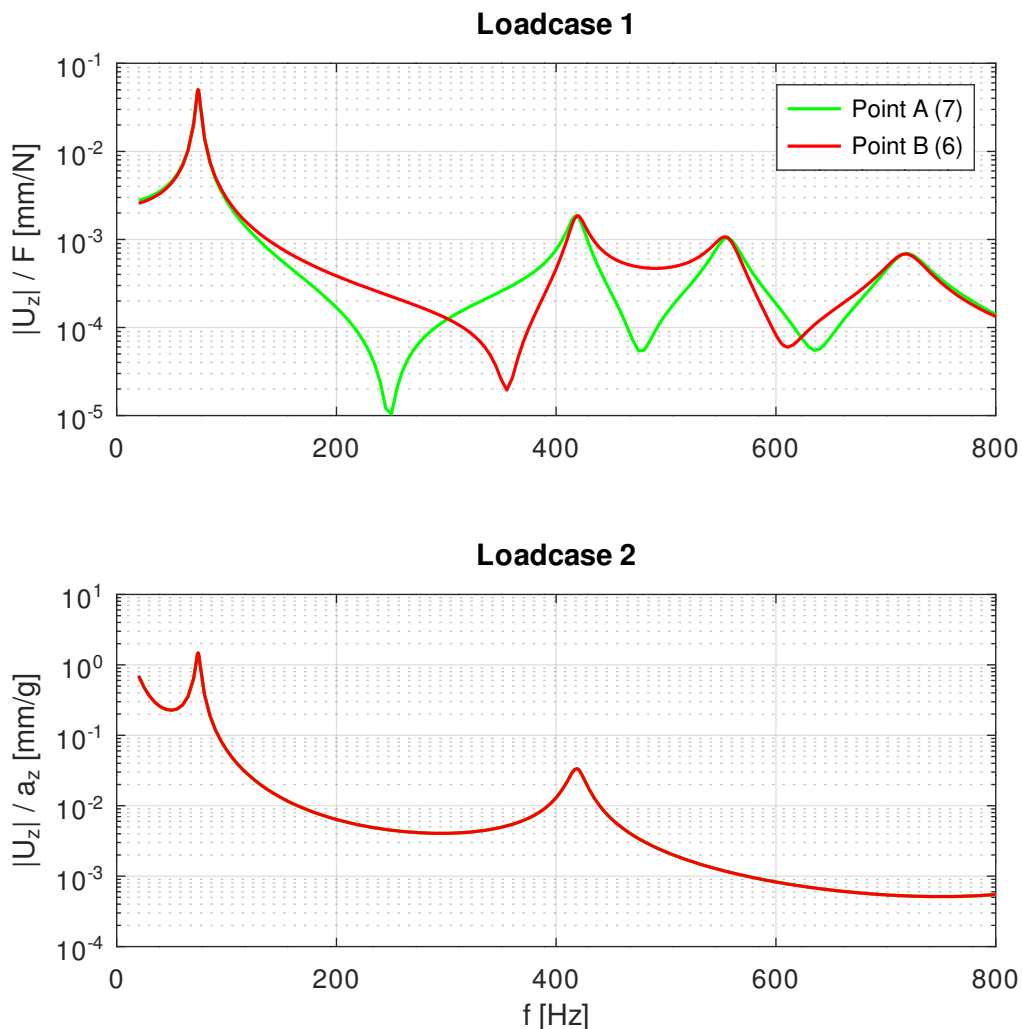


Figure 4.2-7: Box Beam, Transfer Functions for Vertical Displacements

combined in a cell array which is input to function **mfs\_getresp**.

Function **mfs\_getresp** returns a cell array of structures, each cell containing the results of one element. The fields of the structure depend on the element type. Because the local x-axis of the element coordinate system of the elements of interest coincides with the global x-axis, the stresses of interest are those in field **sigxu** ( $\sigma_x$  with respect to the element coordinate system at the upper surface of the shell element). For plotting, the stress components of interest are collected in arrays **sigx1** and **sigx2**, the rows of which correspond to elements and the columns to excitation frequencies.

The element identifiers are included in the legend so that we can check if we get the results of the correct elements. The identifiers are contained in field **id** of the output structure.

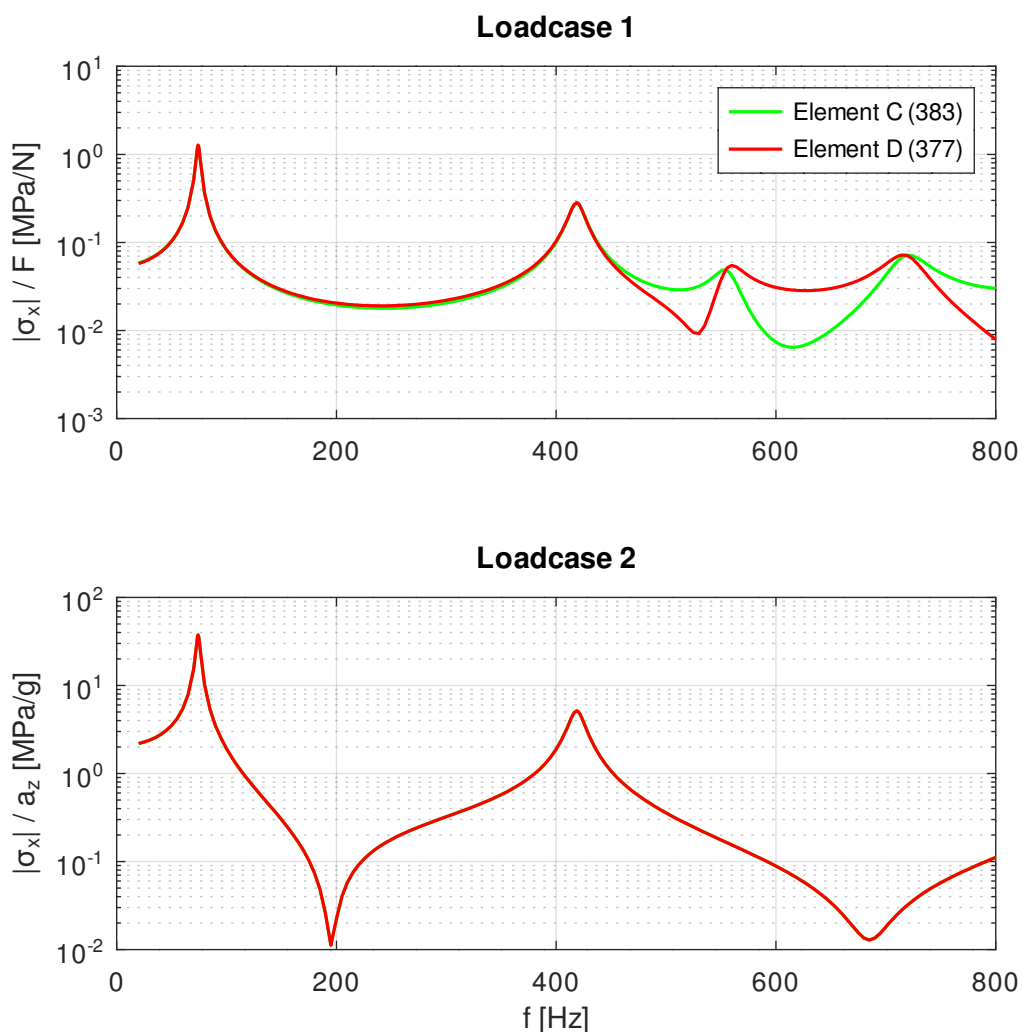


Figure 4.2-8: Box Beam, Transfer Functions for Stresses

```
# Transfer functions for stresses

sig1 = mfs_getresp(beam, "freqresp", "stress", {"C", "D"}, 1);
sig2 = mfs_getresp(beam, "freqresp", "stress", {"C", "D"}, 2);

sigx1 = [sig1{1}.sigxu; sig1{2}.sigxu];
sigx2 = [sig2{1}.sigxu; sig2{2}.sigxu];

lgtext = {sprintf("Element C (%d)", sig1{1}.id), ...
          sprintf("Element D (%d)", sig1{2}.id)};

figure(2, "position", [300, 200, 1000, 750],
       "paperposition", [0, 0, 15, 15]);
subplot(2, 1, 1)
    semilogy(f, abs(sigx1));
    title("Loadcase 1");
    legend(lgtext);
    grid;
    ylabel('| \sigma_x | / F [MPa/N]');
subplot(2, 1, 2)
    semilogy(f, abs(sigx2));
    title("Loadcase 2");
    grid;
    xlabel('f [Hz]');
    ylabel('| \sigma_x | / a_z [MPa/g]');
print("sig.svg", "-dsvg", "-F:10");
```

The transfer functions for the stresses of both load cases are shown in Figure 4.2-8.

Finally, displacements and stresses of the entire model are calculated at the

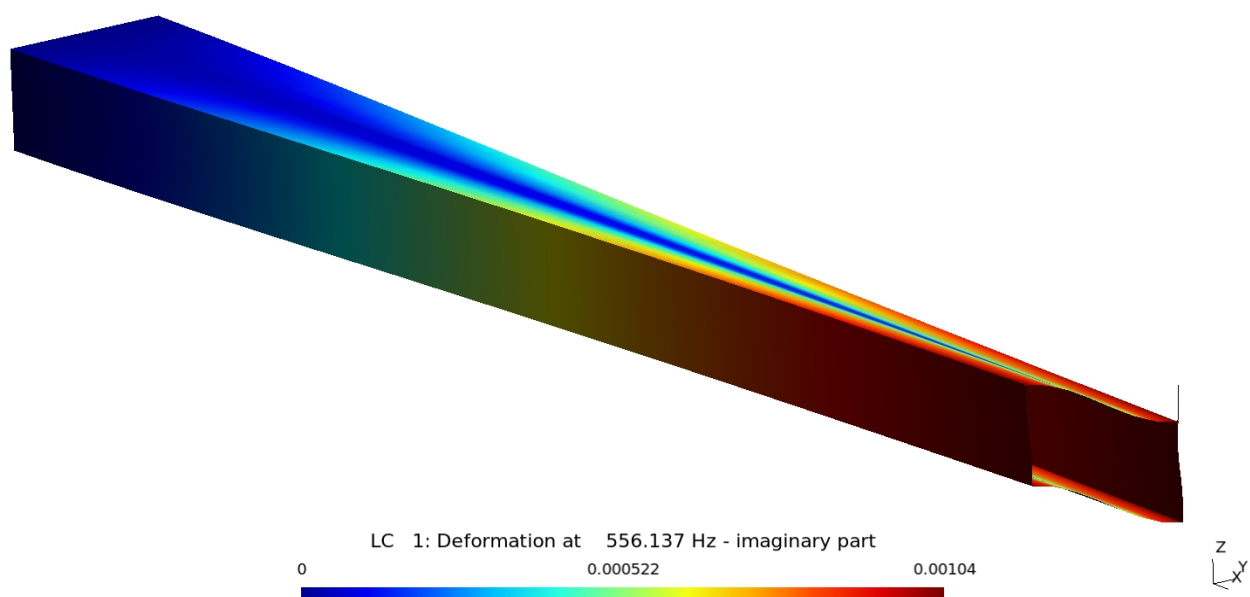


Figure 4.2-9: Box Beam, Load Case 1: Displacements at 556 Hz (Imaginary Part)

peak frequencies of both load cases. Function **mfs\_peaks** is used to find the indices of the peaks of the displacement transfer functions of both load cases. Then function **mfs\_back** performs the back transformation. It is called for each load case to calculate the displacements from the modal coordinates. Subsequently, function **mfs\_results** is called to calculate the element results from the displacements, and function **mfs\_export** to export both displacements and stresses to Gmsh.

```
# Deformation and stress at peak frequencies

ip1 = mfs_peaks(abs(Uz1(1, :)));
ip2 = mfs_peaks(abs(Uz2(1, :)));

beam = mfs_back(beam, "freqresp", "disp", 1, f(ip1));
beam = mfs_back(beam, "freqresp", "disp", 2, f(ip2));
beam = mfs_results(beam, "freqresp", "element");

mfs_export("freq.pos", "msh", beam, "freqresp",
           "disp", "stress");

# Save results

save -binary freqresp.bin beam
```

File **freq.pos**, for each result, load case and excitation frequency, contains

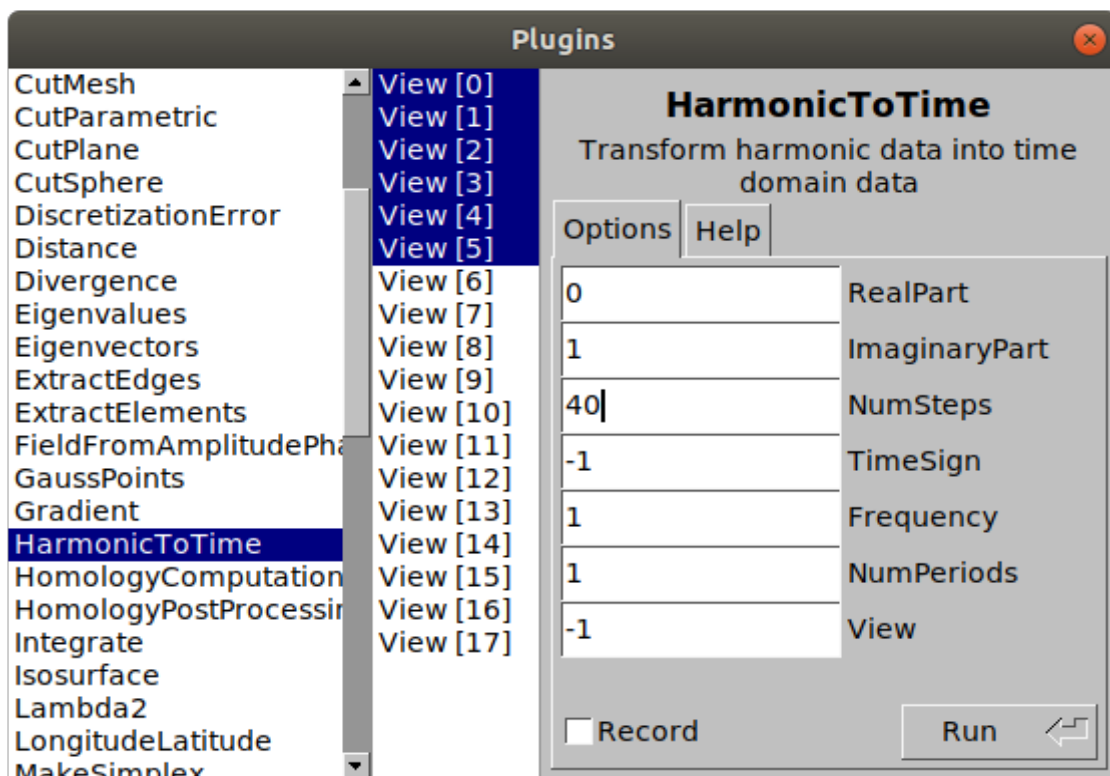


Figure 4.2-10: Gmsh: HarmonicToTime Plugin

two time steps. Time step 0 corresponds to the real part and time step 1 to the imaginary part. Figure 4.2-9, e.g., shows the imaginary part of the displacements of load case 1 at 556 Hz. Using the right and left arrow keys on the keyboard we can switch between real and imaginary part.

Because of

$$u(t) = \Re(U(f)e^{2\pi i f t}) = \Re(U(f))\cos(2\pi f t) - \Im(U(f))\sin(2\pi f t)$$

the imaginary part corresponds to the results at  $t = 3T/4$  where  $T = 1/f$  is the period. Likewise, the real part corresponds to results at  $t = 0$ .

To animate the deformation, we first use the HarmonicToTime plugin (Tools → Plugins) to convert the complex data to the time domain. Variable NumSteps defines the number of pictures per period. We increase this number to 40 to get a smoother and slower animation (see Figure 4.2-10).

For each of the selected views, the plugin creates a new view containing all time steps. After selecting one of these new views, we can use the right and left arrow keys of the keyboard to go through the time steps. Figure 4.2-11 shows the deformation at time step 10. Finally, we can use the animation keys down left on the screen (see Figure 4.2-11) to start and stop the animation.

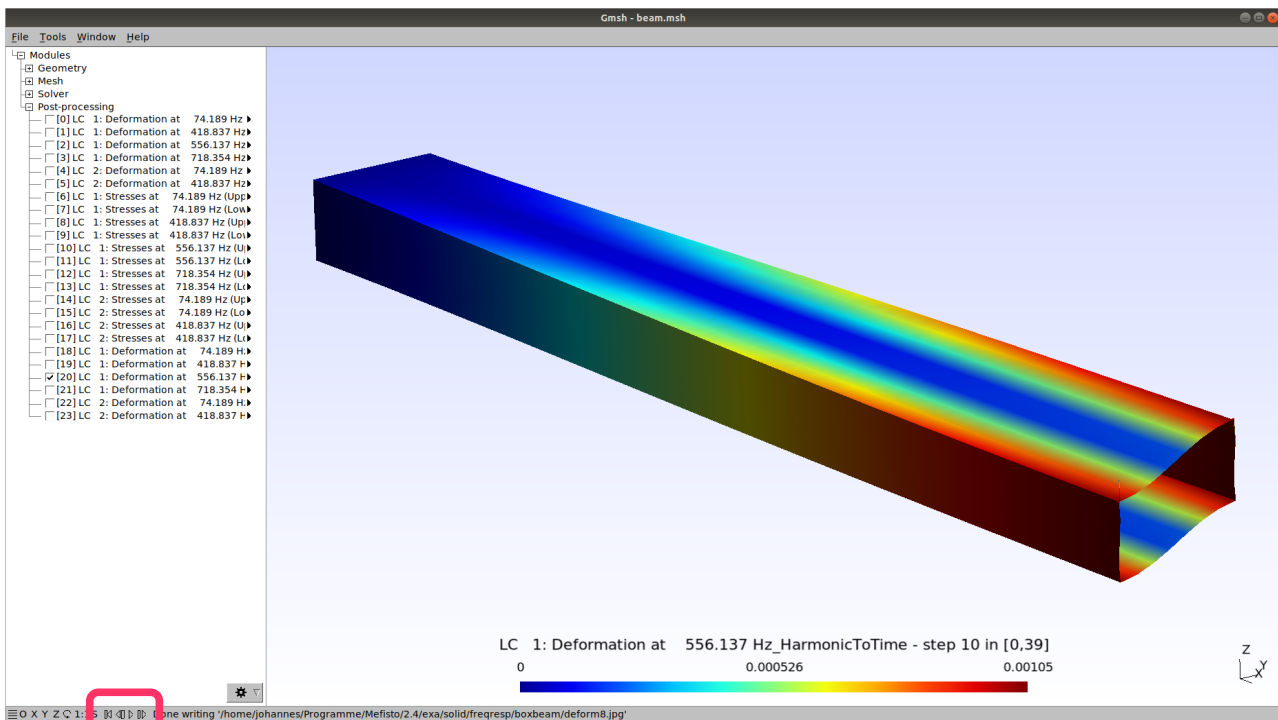


Figure 4.2-11: Box Beam, Deformation at Time Step 10

## Comparison of the Methods and Error Analysis

Script `compare.m` examines how good the error estimate is and how accurate the results of an enhanced modal frequency response analysis are, compared to the results of a direct frequency response analysis and a modal frequency response analysis without static approximation of the neglected modes. This analysis is done for the first load case only.

First, the component with the results of the normal modes analysis is loaded and the elements and the degrees of freedom for output as well as the excitation frequencies are defined. The same excitation frequencies are used for all three methods so that errors can be calculated. The number of excitation frequencies is smaller than in the previous analysis because a direct frequency response analysis is much more time expensive than a modal frequency response analysis. The zero frequency is included because we will also need the static response.

```
%           green      red      blue      magenta
colors = [0, 1, 0; 1, 0, 0; 0, 0, 1; 1, 0, 1];
set(0, "defaultaxescolororder", colors);

fid = fopen("compare.res", "wt");

# Results from normal modes analysis

load modes.bin

# Elements for stress output

elts = {"C", "D"};

# Degrees of freedom for displacements

rid = {"A", 3; "B", 3};

# Excitation frequencies

fmin = 0; % Zero frequency = static solution
df = 10;

fe = fmin : df : fmax; % fmax from modes.bin
```

Next, we repeat the error estimation. This time, however, a list of excitation frequencies is input to function `mfs_reductionerror`, and computations are requested for load case 1 only. The function calculates error estimates at all excitation frequencies and returns them in structure array `rederr`. As only one load case is processed, the structure array contains only one structure whose fields contain the absolute error in strain energy without and with static correction (fields `abs` and `eabs` respectively) and the relative error in strain

energy without and with static correction (fields **rel** and **erel** respectively). The relative errors are obtained by dividing the absolute errors by the static strain energy.

```
# Error estimation

rederr = mfs_reductionerror(fid, beam, fe, 1);
```

Subsequently, a direct frequency response analysis, a modal frequency response analysis and an enhanced modal frequency response analysis are performed. The results are stored in different components: component **beamd** contains the results of the direct frequency response analysis, component **beamm** those of the modal frequency response analysis and component **beame** those of the enhanced modal frequency response analysis.

Please note that the method need be defined for the direct and the modal frequency response analysis, because the default setting is an enhanced modal frequency response analysis.

By default, in a modal frequency response analysis five additional excitation frequencies are added within the halfpower bandwidth of each resonance frequency. As we do not want these frequencies to be added, we explicitly define a number of zero additional frequencies per halfpower bandwidth.

```
# Frequency response analyses

beamd = mfs_freqresp(beam, fe, "loadcase", 1,
                    "method", "direct");
beamm = mfs_freqresp(beam, fe, "nband", 0, "loadcase", 1,
                    "method", "modal");
beame = mfs_freqresp(beam, fe, "nband", 0, "loadcase", 1);
```

Now, function **mfs\_getresp** is used to obtain the results we are interested in, i.e. the strain energy, the vertical displacements at points *A* and *B* and the stresses at points *C* and *D*.

```
# Results

Ed = mfs_getresp(beamd, "freqresp", "strnegy", 1);
Em = mfs_getresp(beamm, "freqresp", "strnegy", 1);
Ee = mfs_getresp(beame, "freqresp", "strnegy", 1);

Ud = mfs_getresp(beamd, "freqresp", "disp", rid, 1);
Um = mfs_getresp(beamm, "freqresp", "disp", rid, 1);
Ue = mfs_getresp(beame, "freqresp", "disp", rid, 1);

sigd = mfs_getresp(beamd, "freqresp", "stress", elts);
sigm = mfs_getresp(beamm, "freqresp", "stress", elts);
sige = mfs_getresp(beame, "freqresp", "stress", elts);
```



```

for n = 1 : length(eltids)
    sigxd(n, :) = sigd{n}.sigxu;
    sigxm(n, :) = sigm{n}.sigxu;
    sigxe(n, :) = sige{n}.sigxu;
endfor

```

The static results are extracted from the first column of the results. They will be used to scale the errors. The strain energies are mean values over one period, i.e. half the amplitude. Therefore, the value obtained must be multiplied by two.

```

# Static results

Es = 2 * Ed(1);
us = abs(Ud(:, 1)); sigxs = abs(sigxd(:, 1));

```

Next, we calculate and plot the relative error in strain energy. This error is the difference between the strain energy calculated with a modal frequency response analysis and the strain energy calculated with a direct frequency response analysis, divided by the strain energy of the static solution. It can be compared with the estimated error in strain energy. At the zero frequency, the error is zero. Therefore, this frequency must be excluded if we want to use a logarithmic scale.

```

# Error in strain energy

```

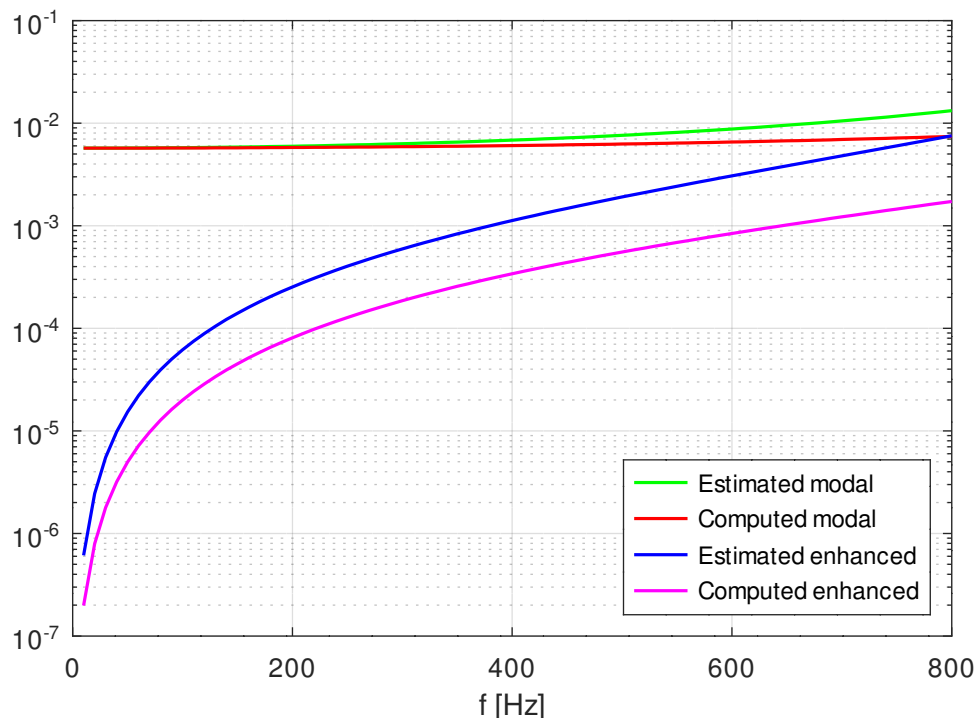


Figure 4.2-12: Box Beam, Estimated and Actual Strain Energy Error



```

em = abs(Em(2 : end) - Ed(2 : end)) / Es;
ee = abs(Ee(2 : end) - Ed(2 : end)) / Es;
f = fe(2 : end);

figure(1, "position", [100, 400, 750, 500],
      "paperposition", [0, 0, 15, 10]);
semilogy(f, rederr.rel(2 : end), f, em,
          f, rederr.erel(2 : end), f, ee);
legend("Estimated modal", "Computed modal",
      "Estimated enhanced", "Computed enhanced",
      "location", "southeast");
grid;
ylim([1e-7, 1e-2]);
xlabel('f [Hz]');
print("see.svg", "-dsvg", "-F:10");

```

Figure 4.2-12 shows the estimated and actual error in strain energy for modal and enhanced modal frequency response analysis. It can be seen that the actual error is less than the estimated error, and that the error is significantly reduced by the static correction.

Next, we calculate and plot the relative displacement error, i.e. the displacement error divided by the static displacement.

```
# Displacement error
```

```

eum = abs(Um(:, 2 : end) - Ud(:, 2 : end)) ./ us;
eue = abs(Ue(:, 2 : end) - Ud(:, 2 : end)) ./ us;

```

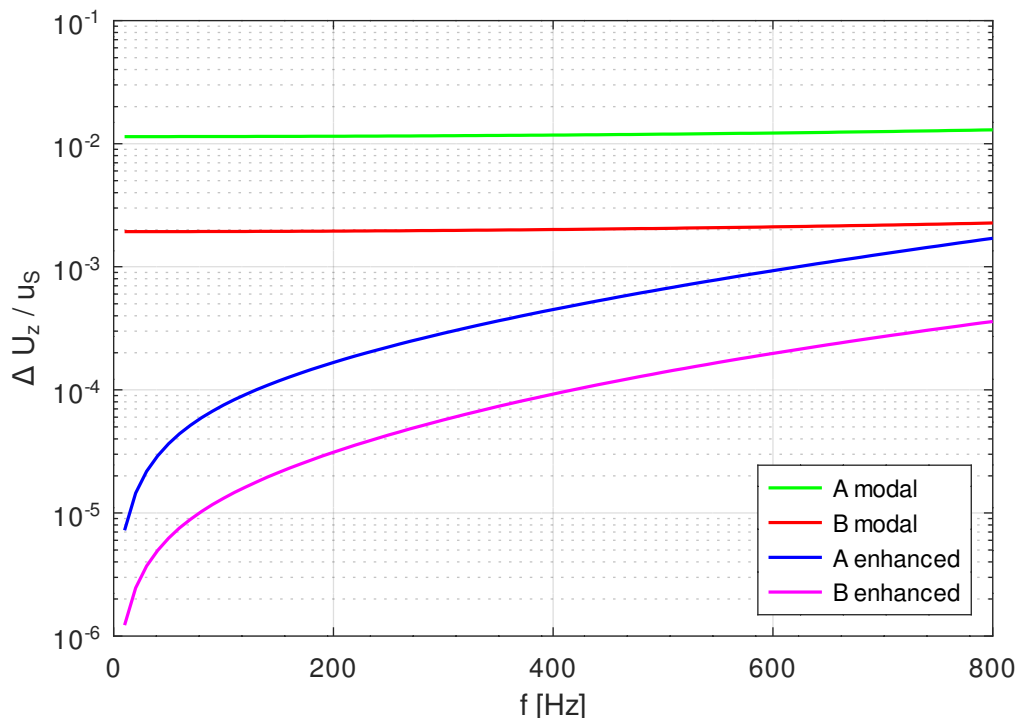


Figure 4.2-13: Box Beam, Displacement Error

```
figure(2, "position", [300, 400, 750, 500],
      "paperposition", [0, 0, 15, 10]);
semilogy(f, eum(1, :), f, eum(2, :),
          f, eue(1, :), f, eue(2, :));
legend("A modal", "B modal",
      "A enhanced", "B enhanced",
      "location", "southeast");
grid;
xlabel('f [Hz]');
ylabel('\Delta U_z / u_S');
print("eu.svg", "-dsvg", "-F:10");
```

Figure 4.2-13 shows that the error of the displacements is very small. Without static correction it is less than 2 % and with static correction it is less than 0.2 % of the static solution.

Finally, we calculate and plot the relative stress error, i.e.

$$\frac{\Delta \sigma_x}{\sigma_{Sx}} = \frac{|\sigma_x^m - \sigma_x^d|}{\sigma_{Sx}}$$

where  $\sigma_{Sx}$  is the static stress.

**# Stress error**

```
esm = abs(sigxm(:, 2 : end) - sigxd(:, 2 : end)) ./ sigxs;
ese = abs(sigxe(:, 2 : end) - sigxd(:, 2 : end)) ./ sigxs;
```

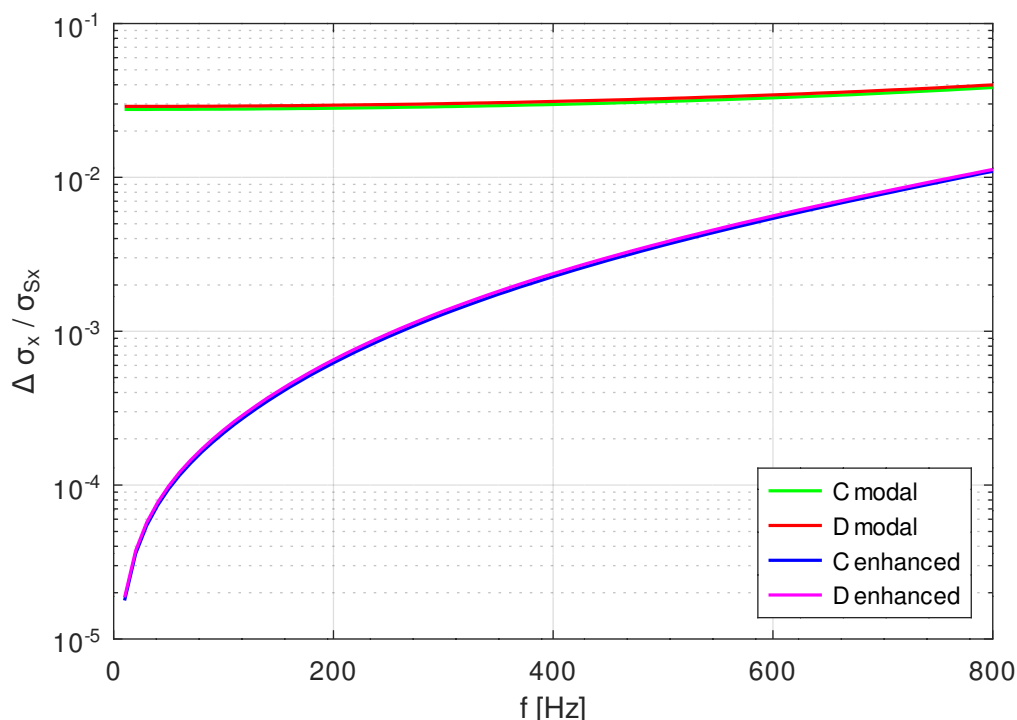


Figure 4.2-14: Box Beam, Stress Error

```

figure(3, "position", [500, 400, 750, 500],
      "paperposition", [0, 0, 15, 10]);
semilogy(f, esm(1, :), f, esm(2, :),
         f, ese(1, :), f, ese(2, :));
legend("C modal", "D modal",
      "C enhanced", "D enhanced",
      "location", "southeast");
grid;
xlabel('f [Hz]');
ylabel('\Delta \sigma_x / \sigma_{Sx}');
print("esig.svg", "-dsvg", "-F:10");

fclose(fid);

```

Figure 4.2-14 shows that the stress error is larger than the displacement error, but with static correction, the error is still less than 2 % of the static solution.

### Modal Contribution Factors

Modal contribution factors describe the contributions of the individual normal modes to the response. The response is the sum of the modal contributions and the static correction. In case of a base motion excitation, the static correction includes the contribution of the base motion.

Function **mfs\_modecont** can be used to calculate the modal contribution factors of displacements. They allow to identify the predominant normal modes.

Script **mcf.m** computes the modal participation factors of the vertical displacements at points *A* and *B* for an excitation frequency of 500 Hz. First, the output file is opened, the excitation frequency and the response identifiers are defined and the transfer functions are loaded.

```

colors = [0, 0, 0; 1, 0, 0; 0, 1, 0; 0, 0, 1; 1, 0, 1;
         1, 0.5, 0];
set(0, "defaultaxescolororder", colors);

fid = fopen("mcf.res", "wt");

freq = 500;
rid = {"A", 3; "B", 3};
...

# Load transfer functions

load freqresp.bin

```

Next, the modal contribution factors are calculated for both load cases and printed in the output file. The optional output argument **res** is structure array

which we will use to plot the modal contribution factors. If the first argument, **fid**, is missing, the contribution factors are only computed but not printed. The last argument is the list of load cases to be processed.

```
# Get modal contribution factors

res = mfs_modecont(fid, beam, rid, freq, 1 : 2);
```

The output file contains the following information:

-----  
Modal contribution factors of component "beam"

Loadcase 1:

Node 7, Dof 3:

Frequency = 500.00 Hz: Resp. = ( 1.294e-04, -5.980e-05)

mode	frequency	absolute		relative	
		real	imag	real	phase
4	556.14 Hz	2.102e-04	-4.009e-05	1.457	14.01
3	418.84 Hz	-1.458e-04	-1.389e-05	-0.888	-149.75
6	718.35 Hz	6.509e-05	-4.356e-06	0.427	20.98
1	74.19 Hz	-5.396e-05	-3.894e-07	-0.343	-154.78
21	Inf Hz	2.947e-05	6.776e-21	0.188	24.81
12	1059.12 Hz	1.375e-05	-5.811e-07	0.089	22.39
8	922.21 Hz	8.729e-06	-4.115e-07	0.057	22.11

Node 6, Dof 3:

Frequency = 500.00 Hz: Resp. = (-4.751e-04, 3.033e-05)

mode	frequency	absolute		relative	
		real	imag	real	phase
4	556.14 Hz	-2.102e-04	4.009e-05	0.446	-7.15
3	418.84 Hz	-1.458e-04	-1.389e-05	0.304	9.09
6	718.35 Hz	-6.509e-05	4.356e-06	0.137	-0.18
1	74.19 Hz	-5.396e-05	-3.894e-07	0.113	4.07
12	1059.12 Hz	-1.375e-05	5.811e-07	0.029	1.23
8	922.21 Hz	8.729e-06	-4.115e-07	-0.018	-179.05

Loadcase 2:

Node 7, Dof 3:

Frequency = 500.00 Hz: Resp. = (-2.190e-03, -2.286e-04)

mode	frequency	absolute		relative	
		real	imag	real	phase
3	418.84 Hz	-2.661e-03	-2.363e-04	1.213	-0.88
1	74.19 Hz	1.582e-03	1.345e-06	-0.715	174.09
21	Inf Hz	-9.940e-04	0.000e+00	0.449	-5.96
8	922.21 Hz	-1.306e-04	6.988e-06	0.059	-9.02

Node 6, Dof 3:

Frequency = 500.00 Hz: Resp. = (-2.190e-03, -2.286e-04)

mode	frequency	absolute		relative	
		real	imag	real	phase
3	418.84 Hz	-2.661e-03	-2.363e-04	1.213	-0.88
1	74.19 Hz	1.582e-03	1.345e-06	-0.715	174.09
21	Inf Hz	-9.940e-04	0.000e+00	0.449	-5.96
8	922.21 Hz	-1.306e-04	6.988e-06	0.059	-9.02

The modal contribution factors are sorted according to descending absolute value of the real part of the relative modal contribution factors. The relative modal contribution factors are the absolute modal contribution factors divided by the response. Their sum, including the static response, equals 1. Thus, the real parts of the relative contribution factors sum up to 1 and the imaginary parts to 0.

The printed output comprises the mode numbers, the resonance frequencies, the real and imaginary part of the absolute contribution factors and the real part and the phase relative to the total response of the relative contribution factors. A value of **inf** for the frequency indicates the static correction.

Modal contribution factors are usually visualized by means of polar plots. Structure array **polar** contains all information needed to create such plots. The elements of this structure array correspond to excitation frequencies. As we have requested the modal contribution factors for one frequency only, the array has only one element.

The following code generates polar plots for the modal contribution factors of the vertical displacements at points A and B for load case 1:

```
# Plot some contribution factors

pol = res(1, 1).polar(1);
figure(1, "position", [100, 300, 800, 800],
      "paperposition", [0, 0, 15, 15]);
polar(pol.theta, pol.rho);
legend(pol.legend, "location", "northeastoutside");
```

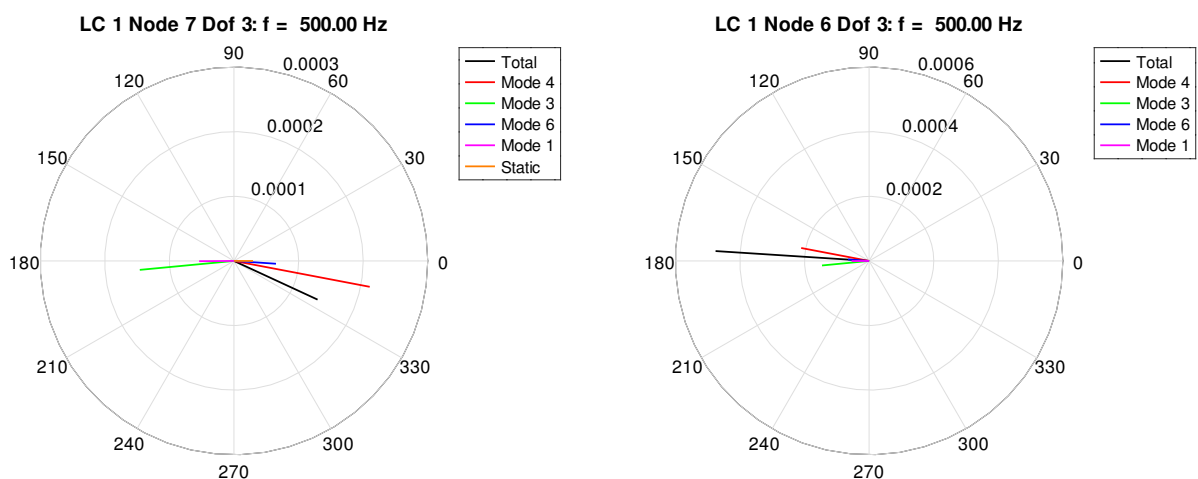


Figure 4.2-15: Box Beam, Modal Participation Factors

```
title(pol.title);  
print(["mcfA", EXT], FORMAT);  
  
pol = res(2, 1).polar(1);  
figure(2, "position", [800, 100, 800, 800],  
       "paperposition", [0, 0, 15, 15]);  
polar(pol.theta, pol.rho);  
legend(pol.legend, "location", "northeastoutside");  
title(pol.title);  
print(["mcfB", EXT], FORMAT);  
  
fclose(fid);
```

Figure 4.2-15 shows the resulting diagrams. It can be seen that the largest contributions come from mode 4 and mode 3. At point *A* (node 7), the contribution of mode 3 reduces the contribution of mode 4 whereas at point *B* (node 6) both contributions are in the same direction. It can also be observed that there is a phase difference of about  $180^\circ$  between the responses at point *A* and point *B*. This is due to the contribution of the antimetric vertical shear mode (mode 4, see Figure 4.2-5).

## 5 Random Response Analysis

### 5.1 Instrument Panel

#### Summary

Directory:	<code>exa/solid/randresp/panel</code>
Objectives:	<ul style="list-style-type: none"> <li>• learn how to perform a random response analysis</li> <li>• learn how to obtain statistical parameters from power spectral densities</li> </ul>
Dimension:	3
Elements:	<code>s4, b2</code>
Constraints:	<code>rigfit</code>
Loads:	prescribed acceleration
Functions:	<code>mfs_beamsection, mfs_import, mfs_new, mfs_stiff, mfs_mass, mfs_massproperties, mfs_freevib, mfs_print, mfs_export, mfs_reductionerror, mfs_freqresp, mfs_getresp, mfs_randresp</code>

#### Problem Description

Figure 5.1-1 shows the structure analysed in this example. It is a simple instrument panel used for the rear seat of a two-seater glider. The panel is made of a sheet of aluminium with a thickness of 1 mm, stiffened by beams on the straight outer edges. The beams have a bar cross section with a width of 1 mm and a height of 5 mm, perpendicular to the panel. They model the folding of the sheet at the edges. The dimensions of the panel can be found in file `panel.geo`.

The instrument panel contains an altimeter, an air-speed indicator and a variometer. The altimeter has a mass of 235 g, the air-speed indicator of 205 g and the variometer of 145 g. The instruments are modelled by point masses that are connected to the four corners of the square enclosing the corresponding circle. The connection is achieved by means of **rigfit** constraints.

The aluminium has a Young's modulus of 70 GPa, a Poisson's ratio of 0.34 and a mass density of 2700 kg/m<sup>3</sup>. The damping is modal damping with a damping ration of 2 % for all normal modes.

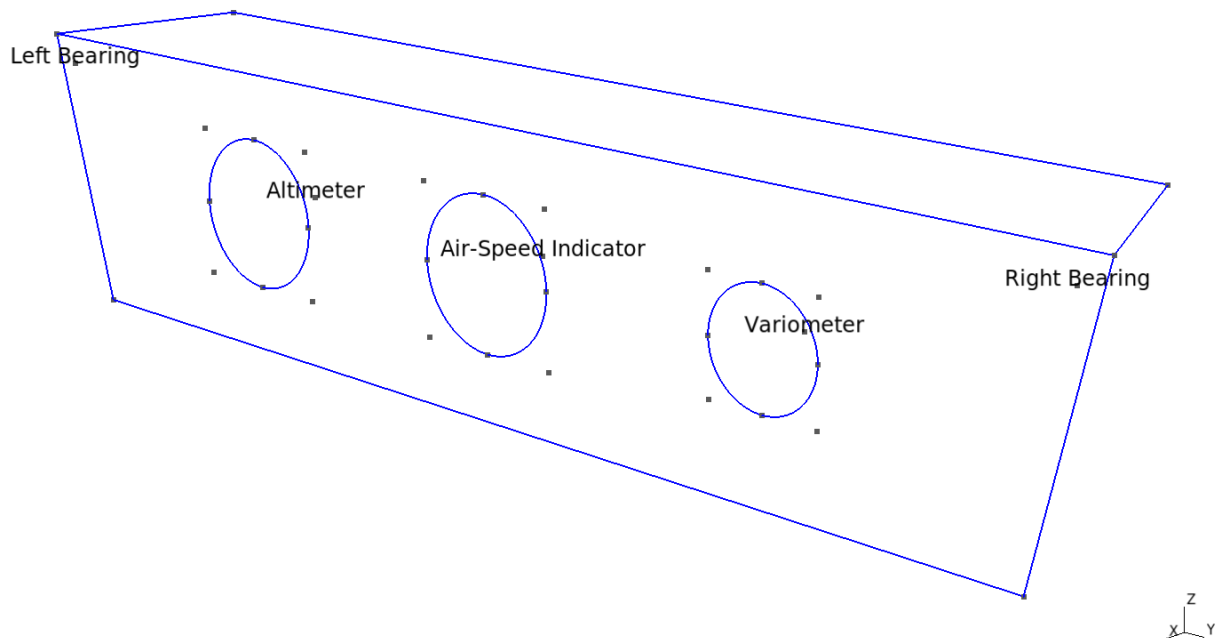


Figure 5.1-1: Instrument Panel

The instrument panel is attached to the aircraft with the bearings at the right and left side. It is excited by prescribed accelerations of the bearings in longitudinal and vertical directions. The lateral accelerations as well as the rotational accelerations are so small that they can be neglected.

The accelerations result from a flight through turbulent air. They are stationary random processes, described by power and cross spectral densities. It is assumed that these are ergodic and Gaussian random processes, an assumption that applies to most technical processes. The power spectral densities are contained in file `Gxx.csv` and the cross spectral densities in file `Gxy.csv`. The units are  $\text{m}^2/(\text{s}^4\text{Hz})$ .

The results to be computed are the power spectral densities of the longitudinal and vertical accelerations of the instruments. Based on the power spectral densities, we determine the RMS values and how often the absolute value of the acceleration exceeds a value of 0.3 g per time. We also determine the probability that the absolute value of the acceleration is greater than 0.3 g.

The power spectral densities of the responses are computed from the power and cross spectral densities of the excitation using the transfer functions. Therefore, the random response analysis is preceded by a frequency response analysis.



## Model Definition

The geometry is defined in file `panel.geo`. It contains the following physical groups:

- **Panel**: used to define the shell elements of the panel
- **Front\_Stiffeners**: used to define the beam elements of the front panel
- **Upper\_Stiffeners**: used to define the beam elements of the upper panel
- **Altimeter**: used to define the point mass of the altimeter
- **AirSpeed**: used to define the point mass of the air-speed indicator
- **Vario**: used to define the point mass of the variometer
- **Bearings**: used to apply the constraints
- **Right\_Ax**, **Right\_Az**, **Left\_Ax**, **Left\_Az**: used to apply the prescribed accelerations in longitudinal and vertical direction at the right and left bearings

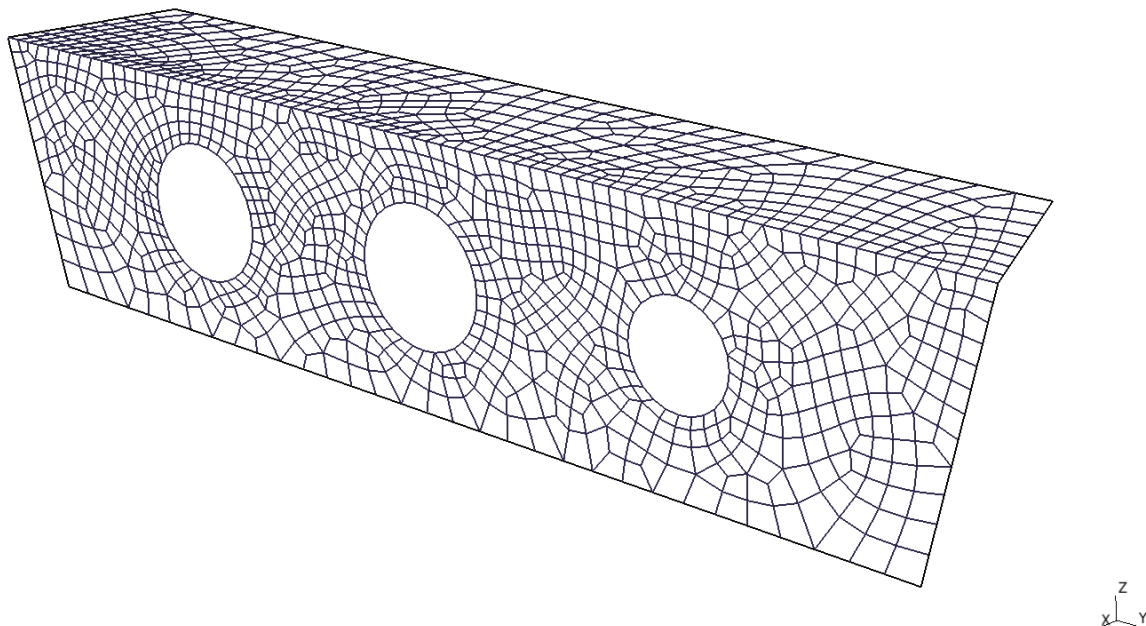


Figure 5.1-2: Instrument Panel, Finite Element Mesh

- **Fix\_Altimeter**: used to attach the altimeter to the panel
- **Fix\_AirSpeed**: used to attach the air-speed indicator to the panel
- **Fix\_Vario**: used to attach the variometer to the panel
- **CM\_Altimeter**: used to define the dependent node of the altimeter
- **CM\_AirSpeed**: used to define the dependent node of the air-speed indicator
- **CM\_Vario**: used to define the dependent node of the variometer

Figure 5.1-2 shows the resulting finite element mesh. The element sizes vary between 10 mm and 20 mm.

The translation data is contained in file `modes.m`. First, the number of normal modes to compute and the damping ratio are defined. Next, the maximum excitation frequency is extracted from the first row of file `Gxx.csv`. The maximum excitation frequency is needed to estimate the error of the modal reduction.

```
fid = fopen("modes.res", "wt");

# Data (N, mm)

% Parameters

nofmod = 3; % Number of normal modes
D      = 0.02; % Modal damping ratio

M      = dlmread("Gxx.csv");
fmax = M(1, end); % Maximum excitation frequency
```

The file is continued with the translation data.

```
% Shell thickness

shell = struct("t", 1);

% Cross section of stiffeners

h = 5; % Height of bar
b = 1; % Width of bar

bar = mfs_beamsection("bar", b, h);

% Material data

mat = struct("type", "iso", "E", 70000, "ny", 0.34,
            "rho", 2.7e-9);

% Masses of instruments
```

```
m_speed = struct("m", 205e-6);
m_alti  = struct("m", 235e-6);
m_vario = struct("m", 145e-6);

# Model definition

data = struct("type", "solid", "subtype", "3d");

% Beam elements

bar.v = [1, 0, 0];
bar.P = [0, 0.5 * h];
data.Front_Stiffeners = struct("type", "elements", "name", "b2",
                              "geom", bar, "mat", mat);

bar.v = [0, 0, 1];
data.Upper_Stiffeners = struct("type", "elements", "name", "b2",
                              "geom", bar, "mat", mat);

% Shell elements

data.Panel = struct("type", "elements", "name", "s4",
                   "geom", shell, "mat", mat);

% Altimeter

data.Altimeter = struct("type", "elements", "name", "m1",
                      "geom", m_alti, "mat", []);

% Air-speed indicator

data.AirSpeed = struct("type", "elements", "name", "m1",
                     "geom", m_speed, "mat", []);

% Variometer

data.Vario = struct("type", "elements", "name", "m1",
                  "geom", m_vario, "mat", []);

% Constraints

data.Bearings = struct("type", "constraints",
                     "name", "prescribed",
                     "dofs", 1 : 6);

% Node sets

data.Fix_Altimeter = struct("type", "nodeset");
data.CM_Altimeter  = struct("type", "nodeset");
data.Fix_AirSpeed  = struct("type", "nodeset");
data.CM_AirSpeed   = struct("type", "nodeset");
data.Fix_Vario     = struct("type", "nodeset");
data.CM_Vario      = struct("type", "nodeset");
```

The load cases are defined as follows:

- Load case 1: Longitudinal acceleration of the right bearing
- Load case 2: Vertical acceleration of the right bearing
- Load case 3: Longitudinal acceleration of the left bearing
- Load case 4: Vertical acceleration of the left bearing

```
% Prescribed accelerations

data.Right_Ax = struct("type", "loads", "name", "acce",
                      "data", [1, 0, 0], "lc", 1);
data.Right_Az = struct("type", "loads", "name", "acce",
                      "data", [0, 0, 1], "lc", 2);
data.Left_Ax  = struct("type", "loads", "name", "acce",
                      "data", [1, 0, 0], "lc", 3);
data.Left_Az  = struct("type", "loads", "name", "acce",
                      "data", [0, 0, 1], "lc", 4);

[model, nset] = mfs_import(fid, "panel.msh", "msh", data);
```

Function **mfs\_import** returns the model definition (variable **model**) and the nodal point sets (variable **nset**). The model definition is completed by adding the definition of the **rigfit** constraints and the damping.

```
% Connect altimeter

dofa = cell(4, 2);
dofa(:, 1) = num2cell(nset.Fix_Altimeter);
dofa(:, 2) = 1 : 3;
model.constraints.rigfit(1) = ...
    struct("nodd", nset.CM_Altimeter, "dofa", {dofa});

% Connect air-speed indicator

dofa(:, 1) = num2cell(nset.Fix_AirSpeed);
dofa(:, 2) = 1 : 3;
model.constraints.rigfit(2) = ...
    struct("nodd", nset.CM_AirSpeed, "dofa", {dofa});

% Connect variometer

dofa(:, 1) = num2cell(nset.Fix_Vario);
dofa(:, 2) = 1 : 3;
model.constraints.rigfit(3) = ...
    struct("nodd", nset.CM_Vario, "dofa", {dofa});

% Damping

model.damping = struct("type", "ratios", "data", D);
```

## Normal Modes Analysis

Finally, the normal modes are computed and upper bounds on the error of the modal reduction are determined for all four load cases.

```
# Analysis

panel = mfs_new(fid, model);
mfs_export("panel.axes", "msh", panel, "mesh", "axes");
mfs_print(fid, panel, "load", "acce");

panel = mfs_stiff(panel);
panel = mfs_mass(panel);
mfs_massproperties(fid, panel);

panel = mfs_freevib(panel, nofmod);
mfs_print(fid, panel, "modes", "freq");
mfs_export("modes.dsp", "msh", panel, "modes", "disp");

mfs_reductionerror(fid, panel, fmax);

# Save results

save -binary modes.bin panel

fclose(fid);
```

The output file `modes.res` contains the excitation, the resonance frequencies, the mass properties and the modal strain energies.

Reading model from file "panel.msh", MSH file version 4.1

Physical Group	Type
Bearings	constraints
Right_Ax	loads
Right_Az	loads
Left_Ax	loads
Left_Az	loads
Fix_Altimeter	nodeset
Altimeter	elements
CM_Altimeter	nodeset
Fix_AirSpeed	nodeset
AirSpeed	elements
CM_AirSpeed	nodeset
Fix_Vario	nodeset
Vario	elements
CM_Vario	nodeset
Panel	elements
Front_Stiffeners	elements
Upper_Stiffeners	elements

Mefisto 2.7: Building new component from input "model"

```
Model Type = solid, Model Subtype = 3d

Number of nodes      = 1407,  Number of elements = 1417
Number of element types = 3
Number of global      degrees of freedom = 8442
Number of local      degrees of freedom = 8412
Number of prescribed degrees of freedom = 12
Number of dependent  degrees of freedom = 18
```

Number of load cases = 4

-----  
Component "panel"

Prescribed accelerations of loadcase 1

node	ax	ay	az	bx	by	bz
7	1.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00

Prescribed accelerations of loadcase 2

node	ax	ay	az	bx	by	bz
7	0.000e+00	0.000e+00	1.000e+00	0.000e+00	0.000e+00	0.000e+00

Prescribed accelerations of loadcase 3

node	ax	ay	az	bx	by	bz
8	1.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00

Prescribed accelerations of loadcase 4

node	ax	ay	az	bx	by	bz
8	0.000e+00	0.000e+00	1.000e+00	0.000e+00	0.000e+00	0.000e+00

The prescribed accelerations are applied correctly. Node 7 is on the right-hand side of the panel and node 8 on the left-hand.

Mass properties of component "panel"

Coordinates of reference point: 0.0000, 0.0000, 0.0000

Rigid body mass matrix:

1.0034e-03	-1.2755e-23	0.0000e+00	-2.6499e-23	-6.1824e-02	1.4081e-02
1.4409e-23	1.0034e-03	1.7403e-25	6.1824e-02	-8.5373e-24	-3.1364e-02
0.0000e+00	-1.7526e-23	1.0034e-03	-1.4081e-02	3.1364e-02	1.7920e-23
-9.3564e-23	6.1824e-02	-1.4081e-02	3.1000e+01	-8.4532e-01	-1.6525e+00
-6.1824e-02	-5.2656e-23	3.1364e-02	-8.4532e-01	6.6677e+00	-1.0562e+00
1.4081e-02	-3.1364e-02	-2.9490e-23	-1.6525e+00	-1.0562e+00	2.7305e+01

Mass = 1.0034e-03

Inertia tensor with respect to reference point:

3.1000e+01	-8.4532e-01	-1.6525e+00
-8.4532e-01	6.6677e+00	-1.0562e+00
-1.6525e+00	-1.0562e+00	2.7305e+01

Coordinates of center of mass: -31.2581, -14.0334, -61.6144

Inertia tensor with respect to center of mass:

2.6993e+01	-4.0517e-01	2.8003e-01
-4.0517e-01	1.8781e+00	-1.8865e-01
2.8003e-01	-1.8865e-01	2.6127e+01

The mass of the instrument panel including the instruments is about 1 kg.

-----  
Component "panel"

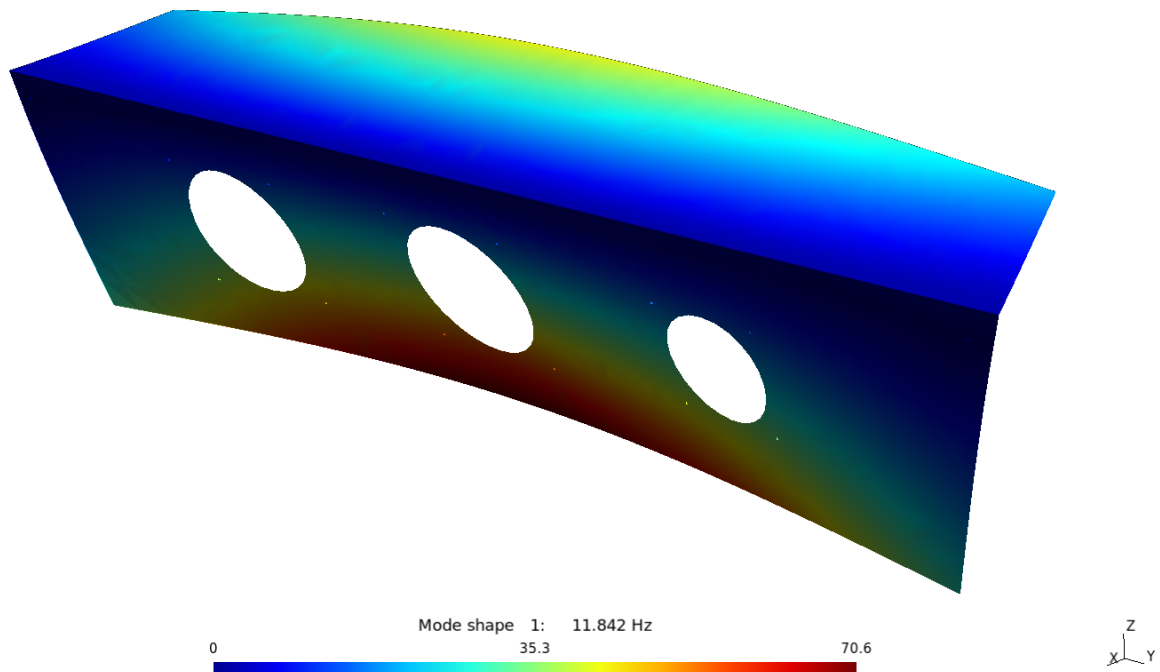


Figure 5.1-3: Instrument Panel, First Normal Mode

Natural frequencies:

Mode	Circ. Frequency	Frequency
1	74.40484	11.84190 Hz
2	179.27467	28.53245 Hz
3	310.50306	49.41810 Hz

All resonance frequencies are above the highest excitation frequency of 10 Hz, so that the transfer functions should only show the effect of the first resonance frequency at 11.8 Hz.

-----  
Modal strain energies of component "panel"

Loadcase 1:

mode	frequency	En/ES	Sum	1 - Sum
1	11.84 Hz	9.48964e-01	0.948964	5.10365e-02
2	28.53 Hz	4.10794e-02	0.990043	9.95703e-03
3	49.42 Hz	2.72268e-06	0.990046	9.95430e-03

Upper bound on relative strain energy error (fmax = 10.00 Hz)  
 with static correction: 4.3408e-04  
 without static correction: 5.4112e-03

Loadcase 2:

mode	frequency	En/ES	Sum	1 - Sum
------	-----------	-------	-----	---------

```

1      11.84 Hz  9.38679e-01  0.938679  6.13211e-02
2      28.53 Hz  1.68033e-02  0.955482  4.45178e-02
3      49.42 Hz  4.87998e-03  0.960362  3.96378e-02

```

```

Upper bound on relative strain energy error (fmax = 10.00 Hz)
with static correction: 1.7285e-03
without static correction: 2.1547e-02

```

Loadcase 3:

mode	frequency	En/ES	Sum	1 - Sum
1	11.84 Hz	9.57423e-01	0.957423	4.25767e-02
2	28.53 Hz	3.20937e-02	0.989517	1.04830e-02
3	49.42 Hz	8.67094e-06	0.989526	1.04743e-02

```

Upper bound on relative strain energy error (fmax = 10.00 Hz)
with static correction: 4.5676e-04
without static correction: 5.6939e-03

```

Loadcase 4:

mode	frequency	En/ES	Sum	1 - Sum
1	11.84 Hz	9.37059e-01	0.937059	6.29408e-02
2	28.53 Hz	2.48746e-02	0.961934	3.80663e-02
3	49.42 Hz	2.14421e-03	0.964078	3.59220e-02

```

Upper bound on relative strain energy error (fmax = 10.00 Hz)
with static correction: 1.5665e-03
without static correction: 1.9527e-02

```

The error analysis shows that the first three normal modes are sufficient. With static correction, the error for all four load cases is less than 0.2 %. Figure 5.1-3 shows the first normal mode which is the most important.

## Frequency Response Analysis

Script **freqresp.m** uses an enhanced modal frequency response analysis to compute the transfer functions between the accelerations of the instruments and the accelerations of the bearings. Each of the four load cases computes the transfer functions of one acceleration.

```

colors = [1, 0, 0; 0, 1, 0; 0, 0, 1];
set(0, "defaultaxescolororder", colors);
...

# Results from normal modes analysis

load modes.bin

# Frequencies

M = dlmread("Gxx.csv");
f = M(1, :);

```

The excitation frequencies are taken from file **Gxx.csv**. Thus, they correspond to the frequencies for which the power and cross spectral densities are



specified, so that no interpolation is required when computing the response spectra. This is possible because the frequency step of 0.1 Hz of these frequencies is sufficient and there are no resonance frequencies below 10 Hz. Function `mfs_freqresp` is prevented from generating additional frequencies in the halfpower bandwidth of the first resonance frequency by setting the value of parameter `nband` to zero.

```
# Frequency response analysis

for lc = 1 : 4
    panel = mfs_freqresp(panel, f, "nband", 0, "loadcase", lc);
endfor

# Plot some transfer functions

rid = {"CM_Altimeter", 1; "CM_AirSpeed", 1;
       "CM_Vario",      1; "CM_Altimeter", 3;
       "CM_AirSpeed",   3; "CM_Vario",    3};

T1 = mfs_getresp(panel, "freqresp", "acce", rid, 1);
T2 = mfs_getresp(panel, "freqresp", "acce", rid, 2);
```

The transfer functions are sorted such that the first 3 functions (rows in matrix **T1** or **T2**) correspond to the longitudinal accelerations and the next 3 functions to the vertical accelerations of the instruments.

```
figure(1, "position", [100, 100, 1000, 500],
```

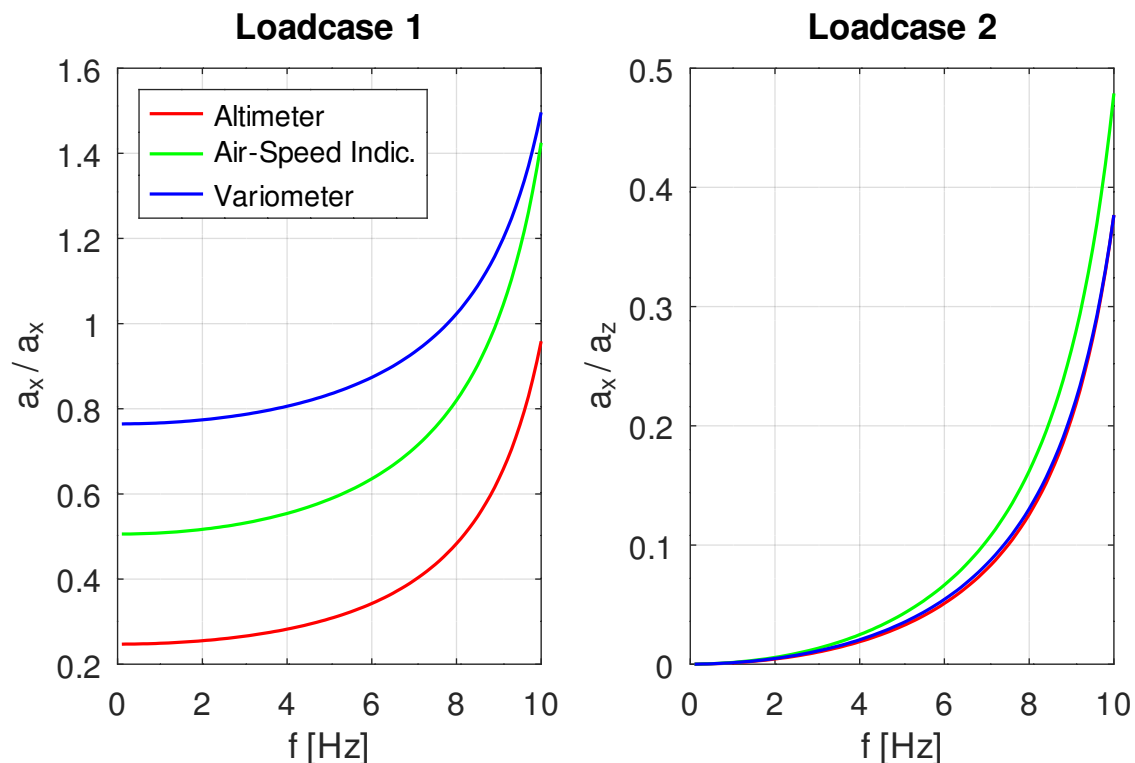


Figure 5.1-4: Instrument Panel, Transfer Functions of Longitudinal Accelerations

```

    "paperposition", [0, 0, 17, 10]);
subplot(1, 2, 1)
    plot(f, abs(T1(1 : 3, :)));
    title("Loadcase 1");
    legend("Altimeter", "Air-Speed Indic.", "Variometer",
        "location", "northwest");
    grid;
    xlabel('f [Hz]'); ylabel('a_x / a_x');
subplot(1, 2, 2);
    plot(f, abs(T2(1 : 3, :)));
    title("Loadcase 2");
    grid;
    xlabel('f [Hz]'); ylabel('a_x / a_z');
print(["ax", EXT], FORMAT);

figure(2, "position", [400, 100, 1000, 500],
    "paperposition", [0, 0, 17, 10]);
subplot(1, 2, 1)
    plot(f, abs(T1(4 : 6, :)));
    title("Loadcase 1");
    legend("Altimeter", "Air-Speed Indic.", "Variometer",
        "location", "northwest");
    grid;
    xlabel('f [Hz]'); ylabel('a_z / a_x');
subplot(1, 2, 2);
    plot(f, abs(T2(4 : 6, :)));
    title("Loadcase 2");
    grid;
    xlabel('f [Hz]'); ylabel('a_z / a_z');
print(["az", EXT], FORMAT);

```

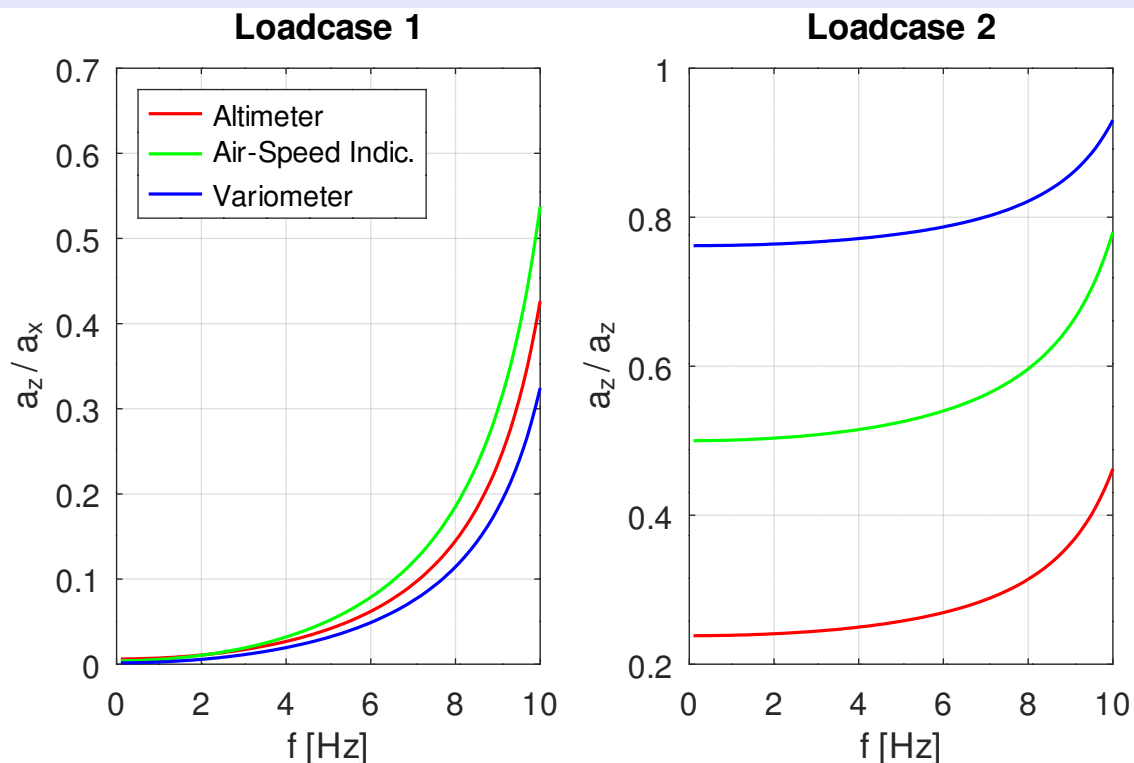


Figure 5.1-5: Instrument Panel, Transfer Functions of Vertical Accelerations

```
# Save results
```

```
save -binary freqresp.bin panel
```

Figure 5.1-4 shows the transfer functions between the longitudinal accelerations of the instruments and the longitudinal (load case 1) and vertical (load case 2) accelerations of the bearing on the right-hand side. Figure 5.1-5 shows the corresponding transfer functions of the vertical accelerations. As both excitation and response are accelerations, the transfer functions are dimensionless. Since there are no resonance frequencies in the frequency range considered, there are no peaks.

### Random Response Analysis

The random response analysis uses the transfer functions to compute the power spectral densities of the requested results from the power and cross power spectral densities of the excitation.

The power spectral densities are contained in file `Gxx.csv`:

- Row 1 contains the frequencies
- Rows 2 and 3 contain the power spectral densities of the longitudinal and vertical accelerations of the right bearing.
- Rows 4 and 5 contain the power spectral densities of the longitudinal

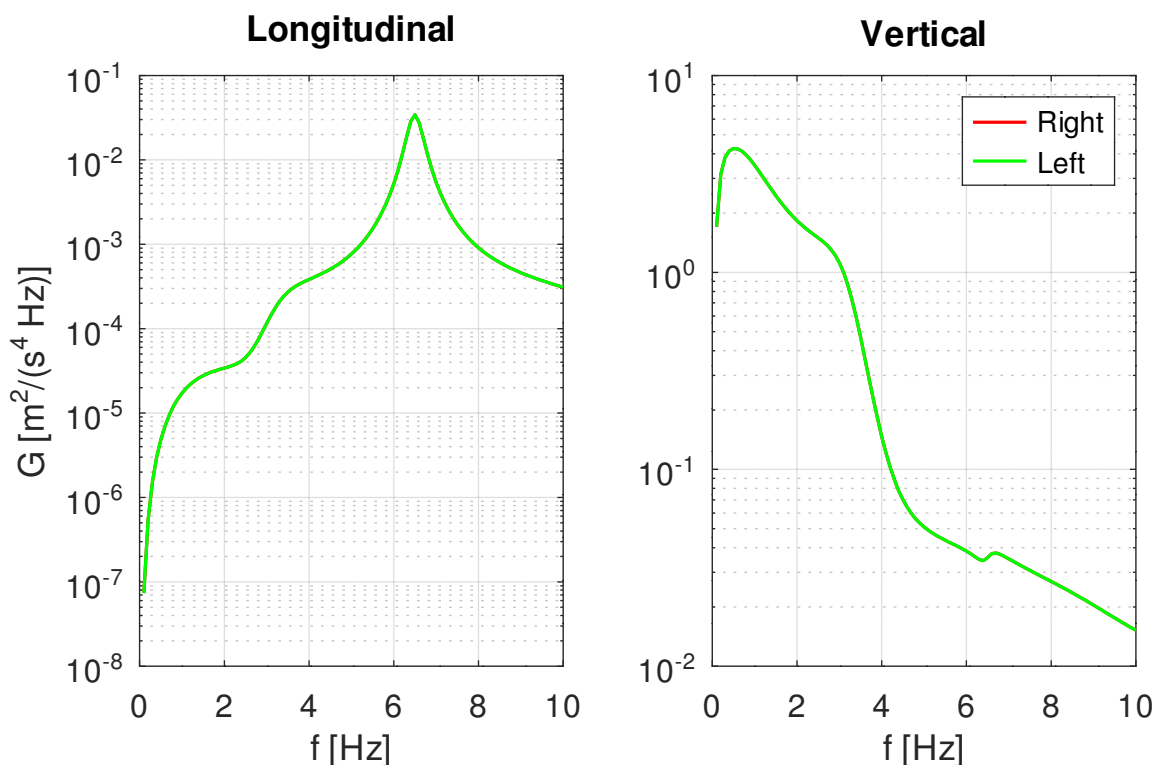


Figure 5.1-6: Instrument Panel, Power Spectral Densities of Excitation

and vertical accelerations of the left bearing.

File `Gxy.csv` contains the cross spectral densities:

- Rows 1 to 3 contain the cross spectral densities between the longitudinal acceleration of the right bearing and the vertical acceleration of the right bearing, the longitudinal acceleration of the left bearing and the vertical acceleration of the left bearing.
- Rows 4 and 5 contain the cross spectral densities between the vertical acceleration of right bearing and the longitudinal and the vertical acceleration of the left bearing.
- Row 6 contains the cross spectral density between the longitudinal and the vertical acceleration of the left bearing.

The unit of all spectral densities is  $\text{m}^2/(\text{s}^4\text{Hz})$ .

Figure 5.1-6 shows the power spectral densities of the excitation. It can be observed that the excitations at the right and left bearings are practically identical and that the excitation in the vertical direction is more than ten times greater than the excitation in the longitudinal direction.

There are two ways to use function `mfs_randresp` to compute the power spectral densities of the responses. In this example we will use both of them.

The first option is to use function `mfs_getresp` to extract the transfer functions and then use function `mfs_randresp`. This method is illustrated by script `randresp1.res`. First, some parameters and data are defined, the results of the frequency response analysis are loaded and the power and cross spectral densities are read from files `Gxx.csv` and `Gxy.csv`.

```
colors = [1, 0, 0; 0, 1, 0; 0, 0, 1];
set(0, "defaultaxescolororder", colors);

# Data

g = 9.81; % Gravity acceleration in m/s^2
a0 = 0.3 * g; % Acceleration level
...

fid = fopen("randresp1.res", "wt");

# Results from frequency response analysis

load freqresp.bin

# Frequencies, power and cross spectral densities

M = dlmread("Gxx.csv");
f = M(1, :);
Gxx = M(2 : end, :);
```

```
Gxy = dlmread("Gxy.csv");
```

Next the transfer matrices are extracted from the component. Function **mfs\_getresp** returns the matrices for each load case. The matrices are collected in cell array **T**. The columns of each matrix correspond to the frequencies and the rows to the responses. The first three rows contain the longitudinal accelerations and the last three rows the vertical accelerations.

```
# Collect transfer functions

rid = {"CM_Altimeter", 1; "CM_AirSpeed", 1;
       "CM_Vario",      1; "CM_Altimeter", 3;
       "CM_AirSpeed",   3; "CM_Vario",     3};

for lc = 1 : 4
    T{lc} = mfs_getresp(panel, "freqresp", "acce", rid, lc);
endfor
```

Function **mfs\_randresp** expects transfer matrices corresponding to one response each, with rows corresponding to load cases. These matrices are collected in cell array **H**. The first three cells contain the matrices corresponding to the longitudinal accelerations and the last three cells the matrices corresponding to the vertical accelerations.

```
for n = 1 : 6
    for lc = 1 : 4
        H{n}(lc, :) = T{lc}(n, :);
    endfor
endfor
```

Now, function **mfs\_randresp** is used to compute the power spectral densities of the responses. Since the transfer matrices between accelerations and accelerations are dimensionless, the dimension of the power spectral density of the response is equal to the dimension of the power spectral density of the excitation. Therefore, no unit conversions are necessary.

Function **mfs\_randresp** requires that the frequencies of the transfer functions match the frequencies of the spectra. This requirement is fulfilled because we computed the transfer functions for the frequencies of the spectra. Otherwise, we would have to use function **interp1** to interpolate the spectra to the frequencies of the transfer functions.

For the computation of power spectral densities only one transfer matrix is needed. In contrast, the transfer matrices of both responses are required to compute the cross spectral density between two responses. Subsequently, the power spectral densities of the excitation and the responses are plotted.

```
# Compute response spectra
```

```

for n = 1 : 6
    G(n, :) = mfs_randresp(Gxx, Gxy, H{n});
endfor

# Plot input spectra

figure(1, "position", [100, 100, 1000, 500],
       "paperposition", [0, 0, 17, 10]);
subplot(1, 2, 1);
    semilogy(f, Gxx([1, 3], :))
    title("Longitudinal");
    grid;
    xlabel('f [Hz]'); ylabel('G [m^2/(s^4 Hz)]');
subplot(1, 2, 2);
    semilogy(f, Gxx([2, 4], :))
    title("Vertical");
    legend("Right", "Left");
    grid;
    xlabel('f [Hz]');
print(["InputG", EXT], FORMAT);

# Plot response spectra

figure(2, "position", [100, 300, 1000, 500],
       "paperposition", [0, 0, 17, 10]);
subplot(1, 2, 1);
    semilogy(f, G(1 : 3, :))
    title("Longitudinal");

```

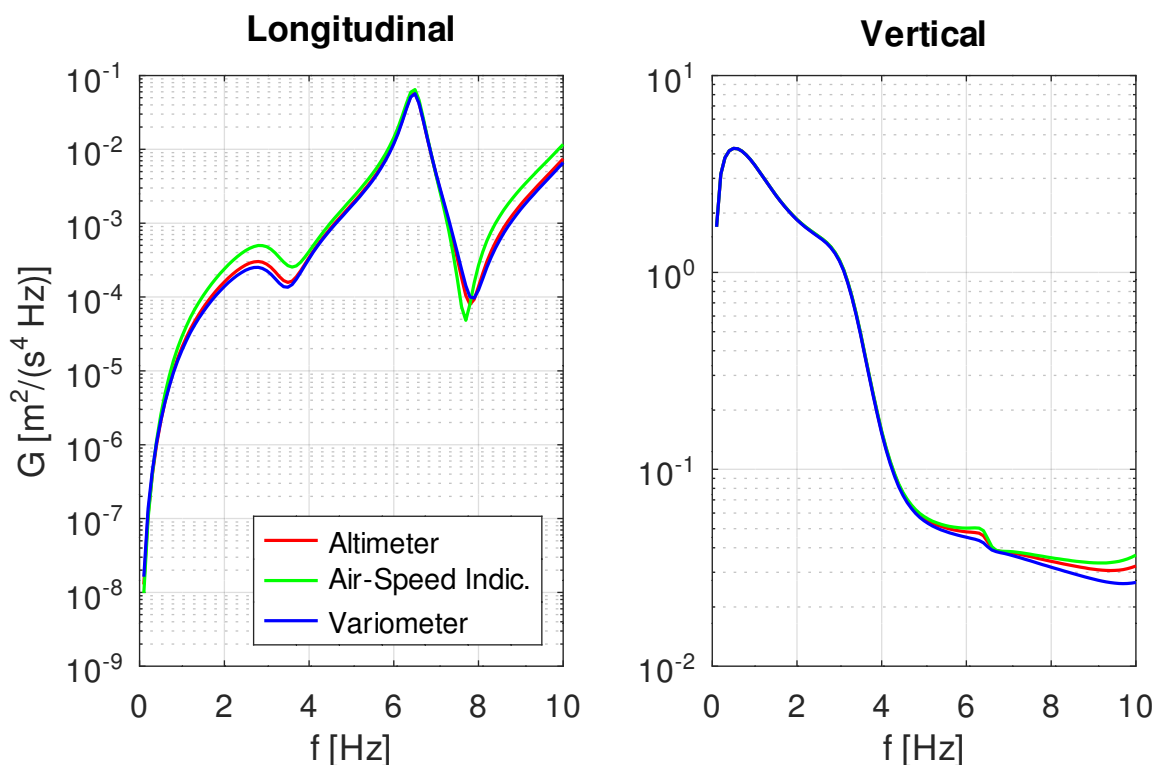


Figure 5.1-7: Instrument Panel, Power Spectral Densities of Response

```

    legend("Altimeter", "Air-Speed Indic.", "Variometer",
           "location", "southeast");
    grid;
    xlabel('f [Hz]'); ylabel('G [m^2/(s^4 Hz)]');
    subplot(1, 2, 2);
    semilogy(f, G(4 : 6, :))
    title("Vertical");
    grid;
    xlabel('f [Hz]');
    print(["ResponseG_1", EXT], FORMAT);

```

Figure 5.1-7 shows the power spectral densities of the longitudinal and vertical accelerations of the instruments. The spectra are quite similar to the excitation spectra. The difference in the power spectral densities of the longitudinal accelerations is due to the transfer functions.

Finally, we compute some statistical parameters. The mean square value  $\psi_a^2$  is the integral of the power spectral density:

$$\psi_a^2 = \int_0^{\infty} G_{aa}(f) df$$

Because the mean value of the random process is zero, the mean square value equals the variance which is the square of the standard deviation  $\sigma_a^2$ :

$$\psi_a^2 = \sigma_a^2$$

The standard deviation is also known as the RMS value (root mean square). We use function **trapz** to perform the integration according to the trapezoidal rule.

A further statistical parameter is the number of positive zero crossings, i.e. zero crossings from negative to positive values. If the random process is Gaussian, the number of positive zero crossings  $N_0$  can be computed from

$$N_0 = \sqrt{\frac{\int_0^{\infty} f^2 G_{aa} df}{\psi_a^2}}.$$

Once the number of zero crossings is known it is easy to compute the number of positive crossings of any other level  $a_0$ :

$$N(a \uparrow a_0) = N_0 \exp\left(-\frac{a_0^2}{2\sigma_a^2}\right)$$

With

$$N(a \downarrow -a_0) = N(a \uparrow a_0)$$

the number of positive crossings of the absolute value is

$$N_a = N(|a| \uparrow a_0) = N(a \uparrow a_0) + N(a \downarrow -a_0) = 2 N_0 \exp\left(-\frac{a_0^2}{2 \sigma_a^2}\right).$$

The probability density of a Gaussian process depends only on the mean value and the standard deviation. If the mean value is zero it reads

$$p(a) = \frac{1}{\sqrt{2\pi}\sigma_a} \exp\left(-\frac{a^2}{2\sigma_a^2}\right).$$

The probability that the magnitude of  $a$  is greater than  $a_0$  is

$$P(|a| > a_0) = 1 - P(|a| < a_0) = 1 - \int_{-a_0}^{a_0} p(a) da = 1 - 2 \int_0^{a_0} p(a) da.$$

To evaluate the integral

$$2 \int_0^{a_0} p(a) da = 2 \int_0^{a_0} \frac{1}{\sqrt{2\pi}\sigma_a} \exp\left(-\frac{a^2}{2\sigma_a^2}\right) da$$

we substitute

$$t = \frac{a}{\sqrt{2}\sigma_a}, \quad a = \sqrt{2}\sigma_a t, \quad da = \sqrt{2}\sigma_a dt, \quad z_0 = \frac{a_0}{\sqrt{2}\sigma_a}$$

and obtain

$$2 \int_0^{a_0} p(a) da = \frac{2}{\sqrt{\pi}} \int_0^{z_0} \exp(-t^2) dt = \operatorname{erf}(z_0) = \operatorname{erf}\left(\frac{a_0}{\sqrt{2}\sigma_a}\right)$$

where  $\operatorname{erf}(z)$  is the error function. With the complementary error function

$$\operatorname{erfc}(z) = 1 - \operatorname{erf}(z)$$

we eventually get

$$P(|a| > a_0) = \operatorname{erfc}\left(\frac{a_0}{\sqrt{2}\sigma_a}\right).$$

The complementary error function is available as function **erfc** in GNU Octave.

```
# Statistical parameters of excitation

psix = trapz(f, Gxx, 2);
RMSx = sqrt(psix);
N0x = sqrt(trapz(f, f.^2 .* Gxx, 2) ./ psix);
arel = a0 ./ RMSx;
expa = exp(-0.5 * arel.^2);
Nax = 2 * expa .* N0x;
Px = 100 * erfc(arel / sqrt(2));
```



```

fprintf(fid, "Statistical Parameters of Excitation\n\n");
fprintf(fid, "          right      left\n");
fprintf(fid, "  -----\n");
fprintf(fid, "    RMS a_x    %7.4f    %7.4f    m/s2\n", RMSx([1, 3]));
fprintf(fid, "    RMS a_z    %7.4f    %7.4f    m/s2\n", RMSx([2, 4]));
...

# Statistical parameters of response

psiy = trapz(f, G, 2);
RMSy = sqrt(psiy);
N0y = sqrt(trapz(f, f.^2 .* G, 2) ./ psiy);
arel = a0 ./ RMSy;
expa = exp(-0.5 * arel.^2);
Nay = 2 * expa .* N0y;
Py = 100 * erfc(arel / sqrt(2));

fprintf(fid, "\nStatistical Parameters of Response\n\n");
fprintf(fid, "          Alti.    Speed    Vario\n");
fprintf(fid, "  -----\n");
fprintf(fid, "    RMS a_x    %7.4f    %7.4f    %7.4f    m/s2\n",
        RMSy(1 : 3));
...

fclose(fid);

```

The output file contains the following results:

#### Statistical Parameters of Excitation

	right	left	
RMS a_x	0.1516	0.1516	m/s <sup>2</sup>
a_z	2.8617	2.8617	m/s <sup>2</sup>
N_0 a_x	6.5684	6.5684	1/s
a_z	1.9571	1.9571	1/s
N_a a_x	0.0000	0.0000	1/s
a_z	2.3066	2.3066	1/s
P a_x	0.0000	0.0000	%
a_z	30.3752	30.3752	%

#### Statistical Parameters of Response

	Alti.	Speed	Vario	
RMS a_x	0.2012	0.2190	0.1976	m/s <sup>2</sup>
a_z	2.8821	2.8862	2.8758	m/s <sup>2</sup>
N_0 a_x	6.7502	6.8787	6.7201	1/s
a_z	2.0408	2.0586	2.0147	1/s
N_a a_x	0.0000	0.0000	0.0000	1/s
a_z	2.4234	2.4481	2.3869	1/s
P a_x	0.0000	0.0000	0.0000	%
a_z	30.7200	30.7880	30.6137	%

It can be observed that the statistical parameters of the response are of the same order of magnitude as those of the excitation. The longitudinal accelerations are small compared to the vertical accelerations.

This method, i.e. the extraction of the transfer functions from the component and the subsequent calculation of the response spectra, is very flexible. It allows to compute both power and cross spectral densities of the responses.

But it is somewhat cumbersome because the transfer functions need to be extracted and sorted. In addition, if the frequencies of the excitation spectra and the transfer functions are not identical, the excitation spectra must be interpolated.

The second option computes the power spectral densities of the responses using the transfer functions stored in the component. This method is illustrated by script `randresp2.res`.

```
...

fid = fopen("randresp2.res", "wt");
# Results from frequency response analysis

load freqresp.bin

# Frequencies, power and cross spectral densities

M = dlmread("Gxx.csv");
f = M(1, :);
Gxx = M(2 : end, :);
Gxy = dlmread("Gxy.csv");

# Compute response spectra

rid = {"CM_Altimeter", 1; "CM_AirSpeed", 1;
       "CM_Vario", 1; "CM_Altimeter", 3;
       "CM_AirSpeed", 3; "CM_Vario", 3};

G = mfs_randresp(panel, Gxx, Gxy, "acce", rid);

# Plot response spectra

figure(1, "position", [100, 300, 1000, 500],
       "paperposition", [0, 0, 17, 10]);
...

fclose(fid);
```

If the first argument of function `mfs_randresp` is a component the transfer functions are taken from the component. If the frequencies of the excitation spectra do not match the frequencies of the transfer functions, these frequencies must be entered as 6<sup>th</sup> argument. The function then performs the interpolations automatically. This method is therefore more convenient, but it cannot be used to compute cross spectral densities between different responses.

## 6 Transient Response Analysis

### 6.1 Plane Plate

#### Summary

Directory:	exa/solid/transresp/plate
Objectives:	<ul style="list-style-type: none"> <li>• learn how to find the cut-off frequency of a transient load</li> <li>• learn how to define dynamic loads</li> <li>• learn how to define initial conditions</li> <li>• learn how to assess the number of normal modes in a modal transient response analysis</li> <li>• learn how to define Rayleigh damping</li> <li>• learn how to run a modal transient response analysis</li> <li>• learn how to run a direct transient response analysis</li> <li>• learn how to retrieve transient responses</li> <li>• learn how to back-transform results from a transient response analysis</li> <li>• learn how to compute transient results from the results of a frequency response analysis</li> </ul>
Dimension:	3
Elements:	<b>s4</b>
Loads:	concentrated nodal point forces
Functions:	<b>mfs_import, mfs_new, mfs_stiff, mfs_mass, mfs_massproperties, mfs_freevib, mfs_print, mfs_export, mfs_reductionerror, mfs_transresp, mfs_getresp, mfs_back, mfs_results, mfs_freqresp, mfs_freq2time</b>

#### Problem Description

In this example, we study the transient response of a plate. The geometry of the plate is shown in Figure 6.1-1. The plate is simply supported along the edges 2-3, 5-6 and 1-8. The remaining edges are free. At points A, B and C, vertical loads are applied. At these points, we also compute the displacements, velocities and accelerations.

Two situations are considered. In the first situation, the plate is initially at rest

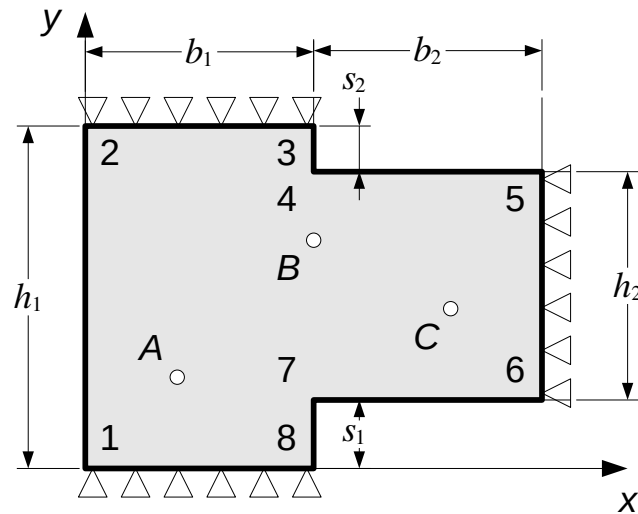


Figure 6.1-1: Plate, Geometry

and undeformed. Then, transient vertical forces are applied at points A, B and C. The transient forces are defined by

$$f_n(t) = F_n \phi_n(t), \quad n \in \{A, B, C\}$$

where  $F_n$  defines the magnitude of the force and

$$\phi_n(t) = \begin{cases} 0, & |t - \Delta t_n| < 0 \\ \sin\left(\pi \frac{t - \Delta t_n}{T}\right), & 0 < |t - \Delta t_n| < T \\ 0, & |t - \Delta t_n| > T \end{cases}$$

the time dependence. These are impulsive loads, with an impulse of

$$\hat{f}_n = \int_0^T f_n(t) dt = \frac{2T}{\pi} F_n.$$

For example, they might describe the impact of some point masses on the plate.

In the second situation, first a static vertical force is applied at point A. The resulting deformation defines the initial condition of a subsequent transient response analysis, i.e. we compute the motion of the plate if the force at point A is suddenly removed.

Geometrical data:

Lengths:

$$h_1 = 300 \text{ mm}, \quad h_2 = 200 \text{ mm}, \quad b_1 = 200 \text{ mm}, \quad b_2 = 200 \text{ mm}, \quad s_1 = 60 \text{ mm}, \quad s_2 = 40 \text{ mm}$$

Thickness:  $t = 2$  mm

Coordinates:

$$A = (80, 80) \text{ mm}, B = (200, 200) \text{ mm}, C = (320, 140) \text{ mm}$$

Material data:

- Young's modulus  $E = 210$  GPa
- Poisson's ratio  $\nu = 0,3$
- Mass density  $\rho = 7850$  kg/m<sup>3</sup>
- Rayleigh damping coefficients  $\alpha_K = 2 \cdot 10^{-5}$  s and  $\alpha_M = 20$  s<sup>-1</sup>

Load data:

Duration of pulse:  $T = 0,01$  s

Magnitude and delay:

	Point A	Point B	Point C	
$F_n$	25	40	50	N
$\Delta t_n$	0	4	2	s
$\hat{f}_n$	0,159	0,255	0,318	Ns

With these data, the mass of the plate is 1,57 kg. An impulse of 0,32 Ns describes, for example, a perfectly elastic collision with a mass of about 16 g and a velocity of 10 m/s.

### Analysis of the Load

A transient response analysis always begins with an analysis of the frequency content of the excitation. We need to determine a reasonable cut-off frequency to find the number of normal modes the finite element model must be capable to represent.

The functions  $\phi_n(t)$  are computed by function **pulse**:

```
function y = pulse(t, params)

# Input    t(:)          Array with time steps
#          params(2)     (1) Duration of pulse
#                               (2) Time delay
#
# Output   y(:)          Time history
#
# -----

n = length(t);
y = zeros(1, n);
```

```

T = params(1); delay = params(2);

ix = find(t >= delay & t <= T + delay);
w = pi / T;
y(ix) = sin(w * (t(ix) - delay));

end

```

The input arguments of this function are an array **t** with the time steps and an array **params** with parameters. In our case, **params(1)** defines the duration  $T$  of the pulse and **params(2)** the delay  $\Delta t$ . The output array **y** contains the values of the function at the time steps defined in **t**. This function will be used by function **mfs\_transresp** when computing the transient response.

The commands to study the excitation functions are contained in file **excitation.m**. First, we load the **signal** package. This package contains function **resample**, which we will use to check the cut-off frequency. Next, we define the parameters. The length  $T_s$  of the time signal determines the frequency resolution of the Fourier transform. The frequency step  $\Delta f$  is related to length of the time signal by

$$\Delta f = 1/T_s.$$

The time step  $\Delta t$  is the inverse of the sampling rate  $f_s$ . According to the Nyquist theorem, the Fourier transform should be zero for frequencies above the Nyquist frequency  $f_a$  which corresponds to half the sampling rate:

$$f_a = \frac{1}{2} f_s = \frac{1}{2} \frac{1}{\Delta t}.$$

With the given data we get a Nyquist frequency of 1000 Hz. Actually, the Fourier transform will not be exactly zero at frequencies above 1000 Hz, but its magnitude should be negligible. Of course, this needs to be checked against the results.

The delay has no effect on the magnitude of the Fourier transform. We use a delay equal to the duration of the pulse to better see the effect of truncating the Fourier transform at the cut-off frequency on the pulse.

Finally, we define a cut-off frequency of 450 Hz. In the following we will check how well the inverse Fourier transform of the truncated Fourier transform matches the original function. In practice, of course, determining a good cut-off frequency is an iterative process.

```

pkg load signal

# Data

T      = 0.01;      % Duration of pulse
Ts     = 10 * T;    % Length of time signal
dt     = T / 20;    % Time step

```

```

delay = 0.01; % Delay
fc     = 450; % Cut-off frequency

```

Now, we compute the function values and the Fourier transform. Function **fft** computes the discrete Fourier transform of a time series. To get the Fourier transform of the excitation function, we have to multiply the values of the discrete Fourier transform by the time step.

The first half of array **Y** contains the values of the Fourier transform corresponding to positive frequencies from 0 Hz to the Nyquist frequency. The values of the second half correspond to negative frequencies. Nevertheless, the frequencies in array **f** range from 0 Hz to the sampling rate. However, we plot the Fourier transform only up to the Nyquist frequency.

```

# Function values

t = 0 : dt : Ts;
N = length(t);
y = pulse(t, [T, delay]);

# Fourier transform

```

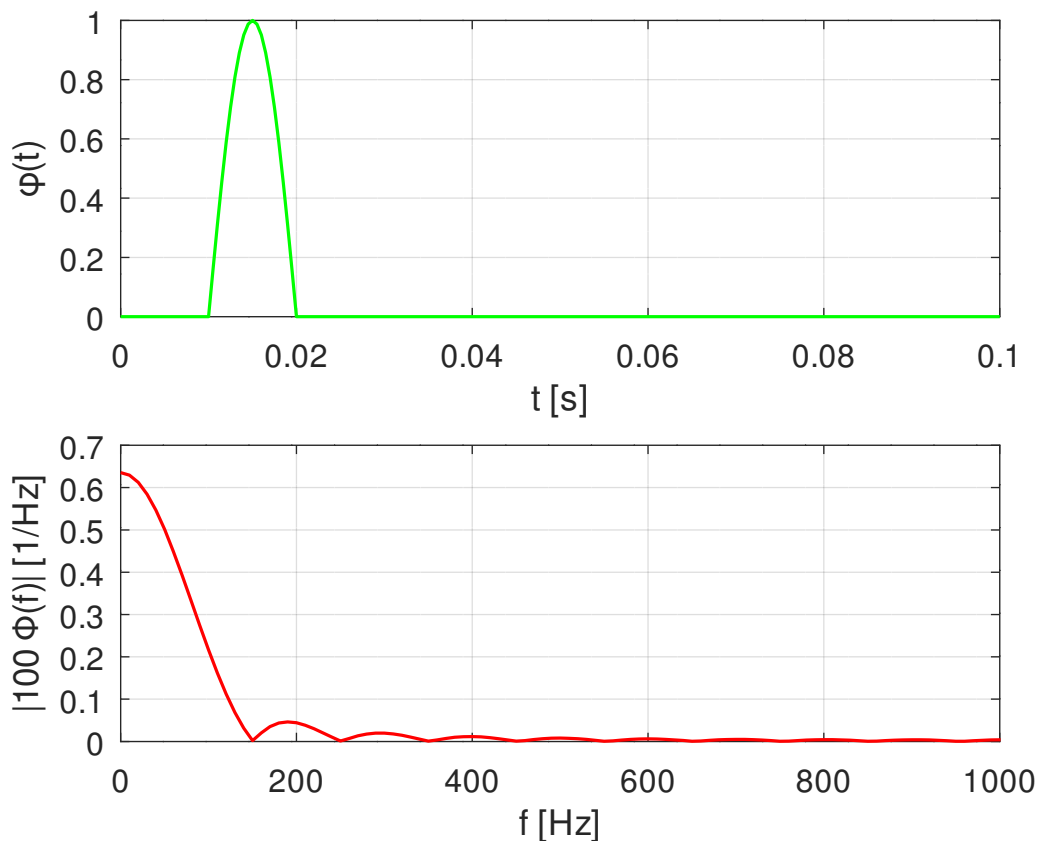


Figure 6.1-2: Plate, Pulse Function and Fourier Transform

```

fs = 1 / dt; fa = 0.5 * fs;
Y = fft(y) * dt;
f = (0 : N - 1) / Ts;

# Plot function and its Fourier transform

figure(1, "position", [100, 100, 750, 750],
      "paperposition", [0, 0, 15, 15]);
subplot(2, 1, 1);
plot(t, y, "color", "green");
grid;
xlabel('t [s]'); ylabel('\phi(t)');
subplot(2, 1, 2);
plot(f, 100 * abs(Y), "color", "red");
grid;
xlim([0, fa]);
xlabel('f [Hz]'); ylabel('|100 \Phi(f)| [1/Hz]');
print("pulse.svg", "-dsvg");

```

Figure 6.1-2 shows the pulse function together with its Fourier transform. It can be seen that the time step is small enough to sample the pulse correctly and that the magnitude of the Fourier transform is very small at frequencies above 450 Hz.

Finally, we investigate the effect of truncating the Fourier transform at the cut-off frequency. For this purpose, we could set the values of the Fourier Transform corresponding to frequencies above the cut-off frequency and below the

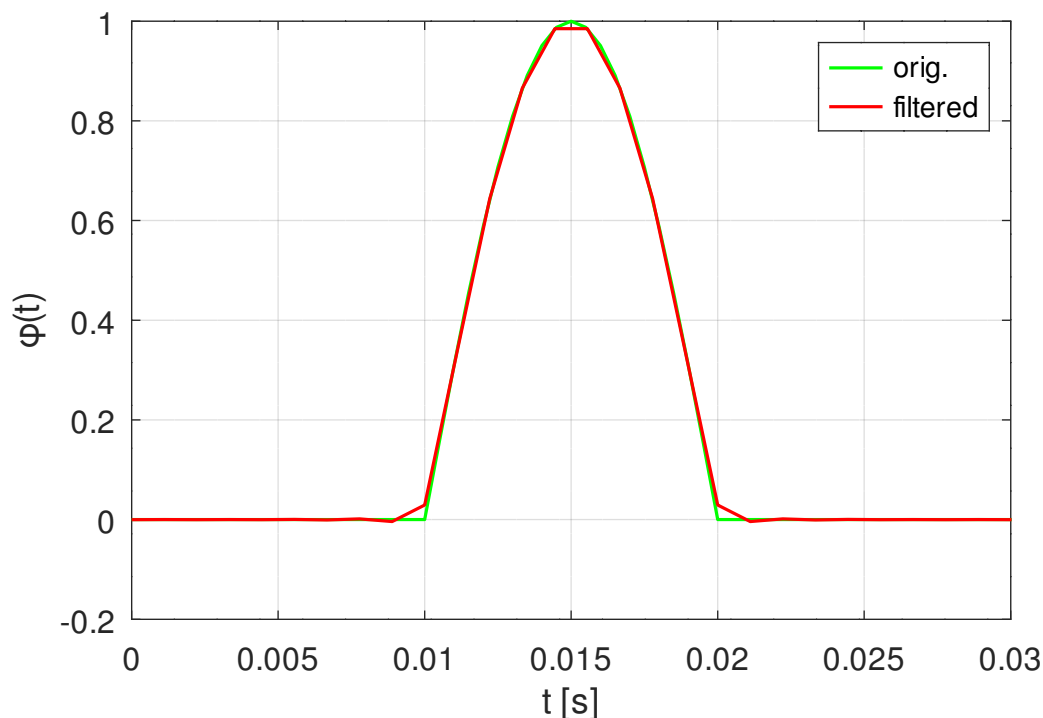


Figure 6.1-3: Plate, Original and Filtered Pulse Function



negative cut-off frequency to zero and subsequently perform the inverse Fourier transformation. However, as these values are stored in the middle of the array **Y**, this is somewhat tricky. It is much easier to use the **resample** function of the **signal** package. This function allows to increase or decrease the sampling rate of a given time series. If the sampling rate is decreased, the time series is filtered to ensure the Nyquist condition is met.

```
# Test of cut-off frequency

yc = resample(y, fc, fa);
tc = (0 : length(yc) - 1) * dt * fa / fc;

figure(2, "position", [500, 100, 750, 500],
        "paperposition", [0, 0, 15, 10]);
plot(t, y, "color", "green", tc, yc, "color", "red");
legend("orig.", "filtered");
grid;
xlim([0, 2 * T + delay]);
xlabel('t [s]'); ylabel('\phi(t)');
print("cutoff.svg", "-dsvg");
```

Figure 6.1-3 shows a good agreement between the original pulse function and the filtered function. Only at the beginning and at the end of the pulse, small deviations can be observed.

### Model Definition

The geometry of the model is defined in Gmsh. The following physical groups are defined:

- **Plate**: shell elements
- **Constraints**: contains the simply supported edges
- **Load\_A**, **Load\_B**, **Load\_C**: used to apply the forces at points *A*, *B* and *C*
- **Resp\_A**, **Resp\_B**, **Resp\_C**: used to obtain the responses at points *A*, *B* and *C*

File **plate.geo** contains the commands to define the geometry and the mesh. An unstructured mesh is used, so only one surface need be defined. The additional points *A* to *C* are assigned to this surface so that they correspond to finite element nodes.

```
// Dimensions

DefineConstant [ h1 = 300,
                  h2 = 200,
                  b1 = 200,
                  b2 = 200,
```

```
s1 = 60,
s2 = 40 ];

DefineConstant [ xA = 80, yA = 80,
                 xB = 200, yB = 200,
                 xC = 320, yC = 140 ];

// Meshing parameters

elen = GetValue("Element Size?", b1 / 20);

Mesh.RecombineAll = 1; Mesh.Smoothing = 1;

// Geometry

Point(1) = {0, 0, 0, elen};
Point(2) = {0, h1, 0, elen};
Point(3) = {b1, h1, 0, elen};
Point(4) = {b1, h1 - s2, 0, elen};
Point(5) = {b1 + b2, h1 - s2, 0, elen};
Point(6) = {b1 + b2, s1, 0, elen};
Point(7) = {b1, s1, 0, elen};
Point(8) = {b1, 0, 0, elen};

Point( 9) = {xA, yA, 0, elen}; // Point A
Point(10) = {xB, yB, 0, elen}; // Point B
Point(11) = {xC, yC, 0, elen}; // Point C

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, 8};
Line(8) = {8, 1};

Curve Loop(1) = {1, 2, 3, 4, 5, 6, 7, 8};
Plane Surface(1) = {1};

Point {9 : 11} In Surface {1};

// Shell elements

Physical Surface("Plate") = {1};

// Constraints

Physical Curve("Constraints") = {2, 5, 8};

// Points with loads

Physical Point("Load_A") = { 9};
Physical Point("Load_B") = {10};
Physical Point("Load_C") = {11};
```

```
// Points for response

Physical Point("Resp_A") = { 9};
Physical Point("Resp_B") = {10};
Physical Point("Resp_C") = {11};

// Mark Points A to C

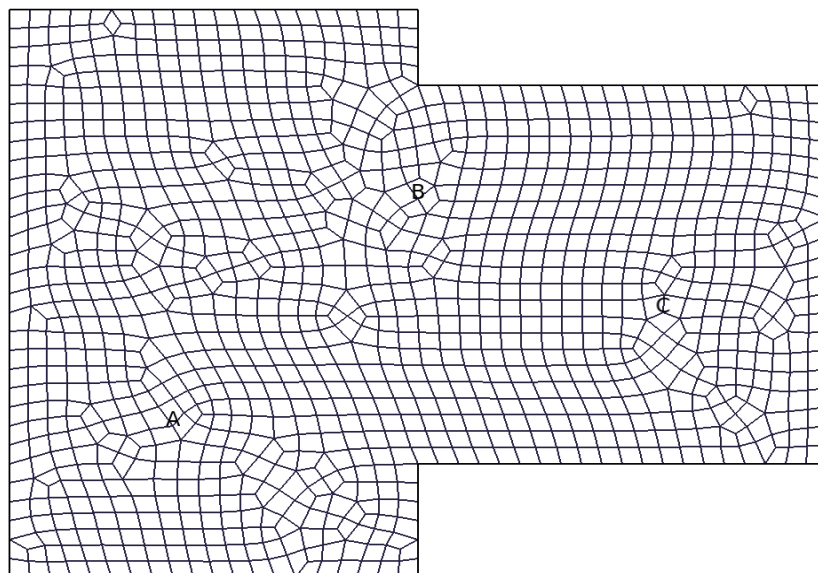
fontsize = 24;
fonttype = 4;
textpos = 1;
font = fontsize + 2^8 * fonttype + 2^16 * textpos;

A = Point{9}; B = Point{10}; C = Point{11};

View "Points" {
  T3(A[0], A[1], A[2], font){ "A" };
  T3(B[0], B[1], B[2], font){ "B" };
  T3(C[0], C[1], C[2], font){ "C" };
};
```

Figure 6.1-4 shows the resulting finite element mesh together with points A to C. The mesh was defined with the proposed default value of the element size.

The translation data are defined in file `plate_model.m`. We define three different load cases. Load case 1 corresponds to a force at point A, load case 2 to a force at point B and load case 3 to a force at point C. The load cases



Y  
|  
Z X

Figure 6.1-4: Plate, Finite Element Mesh

define load patterns that are later linked to time functions.

```
function PLATE = plate_model()

# Example: Plate
#           Model definition
#
# -----

# Data (N, mm):

E   = 210000; % Young's modulus
ny  = 0.3;    % Poisson's ratio
rho = 7.85E-9; % Mass density
t   = 2;      % Thickness

FA  = 25;     % Amplitude of load at point A
FB  = 40;     % Amplitude of load at point B
FC  = 50;     % Amplitude of load at point C

aK  = 2e-5;   % Parameter for Rayleigh-damping
aM  = 20;     % Parameter for Rayleigh-damping

# Material data

mat = struct("type", "iso", "E", E, "ny", ny, "rho", rho);

# Model type and subtype

PLATE = struct("type", "solid", "subtype", "3d");

# Shell elements

geom = struct("t", t);

PLATE.Plate = struct("type", "elements", "name", "s4",
                    "geom", geom, "mat", mat);

# Constraints

PLATE.Constraints = struct("type", "constraints",
                          "name", "prescribed",
                          "dofs", [1 : 3, 6]);

# Damping

PLATE.damping = struct("type", "Rayleigh",
                      "data", [aK, aM]);

# Point loads

PLATE.Load_A = struct("type", "loads", "name", "point",
                    "data", [0, 0, -FA], "lc", 1);
PLATE.Load_B = struct("type", "loads", "name", "point",
                    "data", [0, 0, -FB], "lc", 2);
```

```

    PLATE.Load_C = struct("type", "loads", "name", "point",
                        "data", [0, 0, -FC], "lc", 3);

# Response points

    PLATE.Resp_A.type = "nodeset";
    PLATE.Resp_B.type = "nodeset";
    PLATE.Resp_C.type = "nodeset";

end

```

## Normal Modes Analysis

A transient response analysis always begins with a normal modes analysis. We need to know the resonance frequencies to determine the time step and the simulation time. In addition, we can check whether the discretization is fine enough. Based on the modal static strain energy, we can also check whether the number of normal modes is sufficient for a modal transient response analysis.

File `modes.m` contains the commands to run the normal modes analysis and to compute the modal static strain energies. At the end, the component together with the node sets is stored in file `modes.bin`.

```

# Example: Plate
#           Normal modes and error estimation
#
# -----

    fid = fopen("modes.res", "wt");

# Define simulation parameters

    nofmod = 20; % Number of normal modes
    fmax   = 450; % Maximum excitation frequency

# Define model

    PLATE = plate_model();
    model = mfs_import(fid, "plate.msh", "msh", PLATE);

# Create component and export axes

    plate = mfs_new(fid, model);
    mfs_export("plate.axes", "msh", plate, "mesh", "axes");

# Compute normal modes

    plate = mfs_stiff(plate);
    plate = mfs_mass(plate);

```

```

mfs_massproperties(fid, plate);

plate = mfs_freevib(plate, nofmod);
mfs_print(fid, plate, "modes", "freq");
mfs_export("modes.dsp", "msh", plate, "modes", "disp");

# Estimate reduction error

mfs_reductionerror(fid, plate, fmax);

# Save results

save -binary modes.bin plate

fclose(fid);

```

The output file `modes.res` contains the mass properties, the resonance frequencies and the modal strain energies.

Reading model from file "plate.msh", MSH file version 4.1

Physical Group	Type
Plate	elements
Constraints	constraints
Load_A	loads
Load_B	loads
Load_C	loads
Resp_A	nodeset
Resp_B	nodeset
Resp_C	nodeset

Mefisto 2.7: Building new component from input "model"

```

Model Type = solid, Model Subtype = 3d

Number of nodes      = 1231, Number of elements = 1160
Number of element types = 1
Number of global      degrees of freedom = 7386
Number of local      degrees of freedom = 7134
Number of prescribed degrees of freedom = 252
Number of dependent  degrees of freedom = 0

Number of load cases = 3

```

Mass properties of component "plate"

Coordinates of reference point: 0.0000, 0.0000, 0.0000

Rigid body mass matrix:

1.5700e-03	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00	-2.4178e-01
0.0000e+00	1.5700e-03	0.0000e+00	0.0000e+00	0.0000e+00	2.8260e-01
0.0000e+00	0.0000e+00	1.5700e-03	2.4178e-01	-2.8260e-01	0.0000e+00
0.0000e+00	0.0000e+00	2.4178e-01	4.6452e+01	-4.4273e+01	0.0000e+00
0.0000e+00	0.0000e+00	-2.8260e-01	-4.4273e+01	7.1197e+01	0.0000e+00
-2.4178e-01	2.8260e-01	0.0000e+00	0.0000e+00	0.0000e+00	1.1765e+02

Mass = 1.5700e-03

Inertia tensor with respect to reference point:

4.6452e+01	-4.4273e+01	0.0000e+00
-4.4273e+01	7.1197e+01	0.0000e+00
0.0000e+00	0.0000e+00	1.1765e+02

Coordinates of center of mass: 179.9995, 154.0014, 0.0000

Inertia tensor with respect to center of mass:

```

 9.2172e+00 -7.5263e-01 0.0000e+00
-7.5263e-01 2.0330e+01 0.0000e+00
 0.0000e+00 0.0000e+00 2.9547e+01

```

Component "plate"

Natural frequencies:

Mode	Circ. Frequency	Frequency
1	340.07387	54.12444 Hz
2	585.42220	93.17284 Hz
3	1066.51098	169.74049 Hz
4	1215.02382	193.37705 Hz
5	1476.30496	234.96123 Hz
6	2128.52550	338.76535 Hz
7	2266.63463	360.74611 Hz
8	2605.08661	414.61241 Hz
9	3169.97326	504.51691 Hz
10	3210.70100	510.99894 Hz
11	3672.98746	584.57411 Hz
12	3866.47530	615.36866 Hz
13	4550.47107	724.22996 Hz
14	5002.91527	796.23869 Hz
15	5365.01271	853.86829 Hz
16	5510.26462	876.98585 Hz
17	5678.32211	903.73303 Hz
18	6316.89117	1005.36445 Hz
19	6412.87873	1020.64135 Hz
20	6544.78654	1041.63513 Hz

Modal strain energies of component "plate"

Loadcase 1:

mode	frequency	En/ES	Sum	1 - Sum
1	54.12 Hz	7.05344e-01	0.705344	2.94656e-01
2	93.17 Hz	1.14274e-01	0.819619	1.80381e-01
3	169.74 Hz	1.34872e-02	0.833106	1.66894e-01
4	193.38 Hz	9.06222e-04	0.834012	1.65988e-01
5	234.96 Hz	9.25013e-02	0.926513	7.34866e-02
6	338.77 Hz	1.84782e-05	0.926532	7.34681e-02
7	360.75 Hz	1.37232e-02	0.940255	5.97449e-02
8	414.61 Hz	8.90926e-06	0.940264	5.97360e-02
9	504.52 Hz	1.65172e-02	0.956781	4.32188e-02
10	511.00 Hz	1.11159e-03	0.957893	4.21072e-02
11	584.57 Hz	5.75479e-03	0.963648	3.63524e-02
12	615.37 Hz	8.34150e-04	0.964482	3.55183e-02
13	724.23 Hz	1.04278e-02	0.974910	2.50904e-02
14	796.24 Hz	5.86605e-04	0.975496	2.45038e-02
15	853.87 Hz	4.13892e-04	0.975910	2.40899e-02
16	876.99 Hz	7.80523e-04	0.976691	2.33094e-02
17	903.73 Hz	3.44297e-05	0.976725	2.32750e-02
18	1005.36 Hz	3.02649e-03	0.979752	2.02485e-02
19	1020.64 Hz	1.58531e-03	0.981337	1.86632e-02
20	1041.64 Hz	5.29125e-05	0.981390	1.86103e-02

Upper bound on relative strain energy error (fmax = 450.00 Hz)  
 with static correction: 4.7603e-03  
 without static correction: 1.4065e-02

Loadcase 2:

mode	frequency	En/ES	Sum	1 - Sum
------	-----------	-------	-----	---------

1	54.12 Hz	6.89211e-01	0.689211	3.10789e-01
2	93.17 Hz	1.19378e-01	0.808589	1.91411e-01
3	169.74 Hz	4.02211e-02	0.848810	1.51190e-01
4	193.38 Hz	6.14252e-02	0.910235	8.97646e-02
5	234.96 Hz	2.53470e-03	0.912770	8.72299e-02
6	338.77 Hz	3.16890e-02	0.944459	5.55410e-02
7	360.75 Hz	1.27019e-02	0.957161	4.28391e-02
8	414.61 Hz	1.35454e-04	0.957296	4.27036e-02
9	504.52 Hz	7.20653e-04	0.958017	4.19830e-02
10	511.00 Hz	2.00153e-03	0.960019	3.99814e-02
11	584.57 Hz	4.34999e-03	0.964369	3.56314e-02
12	615.37 Hz	1.95837e-04	0.964564	3.54356e-02
13	724.23 Hz	3.91295e-03	0.968477	3.15227e-02
14	796.24 Hz	4.11317e-04	0.968889	3.11113e-02
15	853.87 Hz	4.62183e-03	0.973510	2.64895e-02
16	876.99 Hz	2.81146e-03	0.976322	2.36781e-02
17	903.73 Hz	2.12360e-04	0.976534	2.34657e-02
18	1005.36 Hz	4.10509e-04	0.976945	2.30552e-02
19	1020.64 Hz	1.33080e-04	0.977078	2.29221e-02
20	1041.64 Hz	3.44164e-03	0.980520	1.94805e-02

Upper bound on relative strain energy error (fmax = 450.00 Hz)  
 with static correction: 4.9828e-03  
 without static correction: 1.4723e-02

Loadcase 3:

mode	frequency	En/ES	Sum	1 - Sum
1	54.12 Hz	3.76313e-01	0.376313	6.23687e-01
2	93.17 Hz	3.94468e-01	0.770781	2.29219e-01
3	169.74 Hz	4.64054e-03	0.775422	2.24578e-01
4	193.38 Hz	1.02888e-01	0.878310	1.21690e-01
5	234.96 Hz	3.34021e-03	0.881650	1.18350e-01
6	338.77 Hz	3.95939e-03	0.885610	1.14390e-01
7	360.75 Hz	1.34016e-02	0.899011	1.00989e-01
8	414.61 Hz	4.39829e-02	0.942994	5.70060e-02
9	504.52 Hz	6.10006e-03	0.949094	5.09059e-02
10	511.00 Hz	1.29036e-04	0.949223	5.07769e-02
11	584.57 Hz	3.39101e-04	0.949562	5.04378e-02
12	615.37 Hz	9.86423e-03	0.959426	4.05735e-02
13	724.23 Hz	4.89381e-04	0.959916	4.00841e-02
14	796.24 Hz	5.06931e-03	0.964985	3.50148e-02
15	853.87 Hz	3.21224e-03	0.968197	3.18026e-02
16	876.99 Hz	3.57101e-04	0.968555	3.14455e-02
17	903.73 Hz	2.69557e-03	0.971250	2.87499e-02
18	1005.36 Hz	6.75553e-04	0.971926	2.80744e-02
19	1020.64 Hz	1.15585e-03	0.973081	2.69185e-02
20	1041.64 Hz	1.62225e-03	0.974704	2.52963e-02

Upper bound on relative strain energy error (fmax = 450.00 Hz)  
 with static correction: 6.4705e-03  
 without static correction: 1.9119e-02

With static correction, the relative error in strain energy is less than 0.65 % for all load cases. The most important normal modes are modes 1 and 2 for all load cases and mode 4 for load case 3. These modes together with mode 20 are shown in Figure 6.1-5. It can be seen that the discretization is fine enough to represent the mode shapes.

## Transient Response Analysis

First, we study the situation of the plate being initially at rest subjected to impact loads. We compare the results of an enhanced modal transient response analysis with those of a direct transient response analysis.



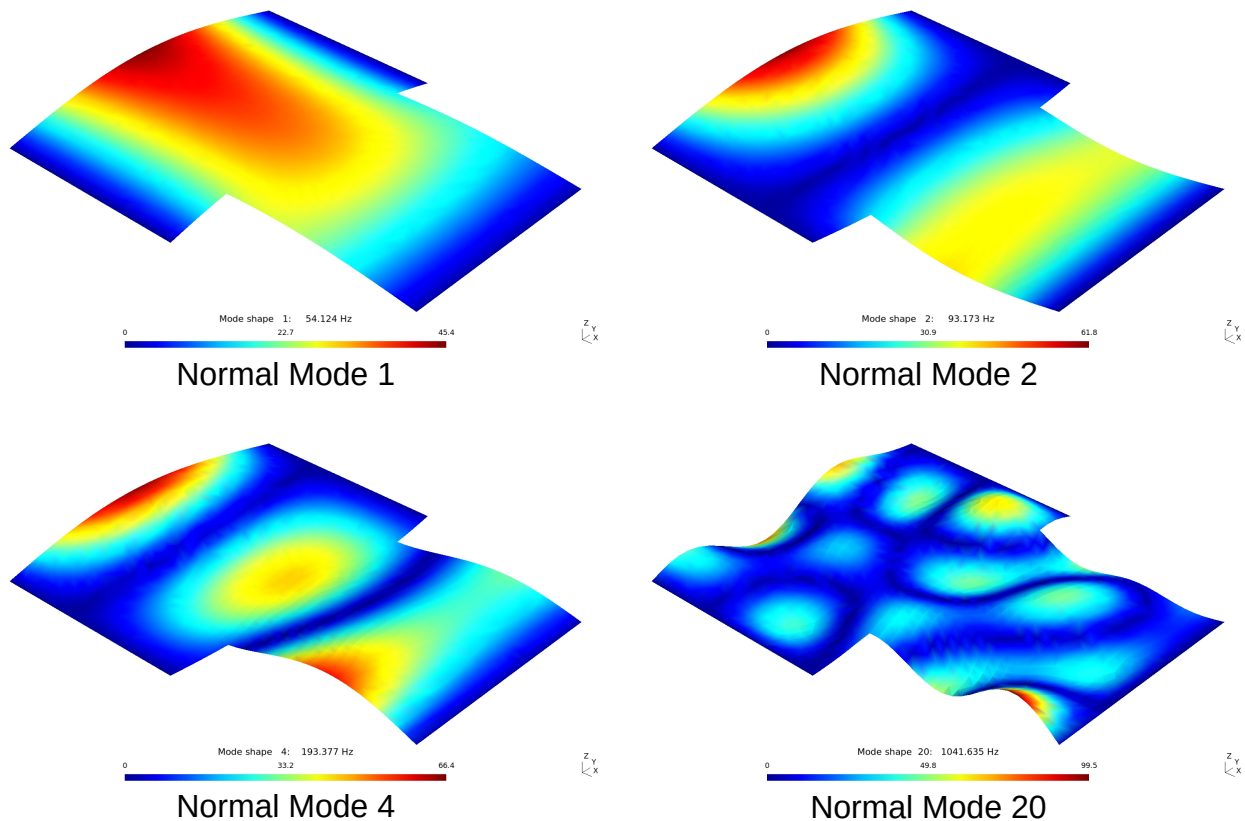


Figure 6.1-5: Some Normal Modes

The commands to run the enhanced modal transient response analysis are contained in file `tresp_impact_modal.m`. The file begins with the definition of the excitation parameters, i.e. the duration of the pulse and the delays at points A, B and C.

```
# Define excitation parameters

T      =      0.01;    % Duration of pulse
delay = [0, 4, 2] * T; % Delays
```

Next we define the time step  $\Delta t$  and the simulation time  $T_s$ . The time step depends on the shortest period which is the inverse of the largest resonance frequency:

$$T_{min} = \frac{1}{f_{max}} = \frac{1}{1042 \text{ Hz}} = 9.60 \cdot 10^{-4} \text{ s}$$

We select a time step  $\Delta t = 10^{-4} \text{ s}$ , so the shortest period is subdivided into about 10 intervals. The simulation time is based on the longest period which is the inverse of the first resonance frequency:

$$T_{max} = \frac{1}{f_1} = \frac{1}{54.12 \text{ Hz}} = 0.0185 \text{ s}$$

We select a simulation time of 0.15 s which includes at least eight periods of the first normal mode.

```
# Define simulation paramters

dt      = 1e-4; % Time step
Ts      = 0.15; % Simulation time
```

Then, we define structure array **loads** that links the load patterns with the time functions. Field **lc** contains the number of the load case defining the pattern, field **func** the user-defined time function and field **params** parameters required by the time function. The time function for all three patterns is function **pulse** already used in the analysis of the load. The parameters of this function are the duration of the pulse and the delay, both defined above.

```
# Define excitation

for l = 1 : 3
    loads(l) = struct("lc", l, "func", "pulse",
                     "params", [T, delay(l)]);
end
```

Now we load the component containing the results of the normal modes analysis and perform the transient response analysis. The second input argument defines the time step and the simulation time. The keyword **"load"** indicates that the following argument is the structure array that links the load patterns to the time functions. The default method is an enhanced modal transient analysis, so no method needs to be specified.

```
# Get results from normal modes analysis

load modes.bin

# Perform transient response analysis

plate = mfs_transresp(plate, [dt, Ts], "load", loads);
```

The results of the transient response analysis are now stored in the component and can be accessed for plotting.

```
# Plot results

rid = {"Resp_A", 3; "Resp_B", 3; "Resp_C", 3};

t = mfs_getresp(plate, "transresp", "time");
w = mfs_getresp(plate, "transresp", "disp", rid);
v = mfs_getresp(plate, "transresp", "velo", rid) / 1000;
a = mfs_getresp(plate, "transresp", "acce", rid) / 9810;

figure(1, "position", [100, 200, 1000, 750],
```

```

        "paperposition", [0, 0, 15, 12]);
plot(t, w);
legend("Pnt. A", "Pnt. B", "Pnt. C", "location", "southeast");
legend("autoupdate", "off");
grid;
xlim([0, Ts]);
line([delay(2), delay(2)], ylim(), "color", "black");
line([delay(3), delay(3)], ylim(), "color", "black");
xlabel('t [s]'); ylabel('w [mm]');

print("impact_modal_w.svg", "-dsvg");

figure(2, "position", [300, 200, 1000, 750],
        "paperposition", [0, 0, 15, 12]);
plot(t, v);
legend("Pnt. A", "Pnt. B", "Pnt. C");
legend("autoupdate", "off");
grid;
xlim([0, Ts]);
line([delay(2), delay(2)], ylim(), "color", "black");
line([delay(3), delay(3)], ylim(), "color", "black");
xlabel('t [s]'); ylabel('v [m/s]');

print("impact_modal_v.svg", "-dsvg");

```

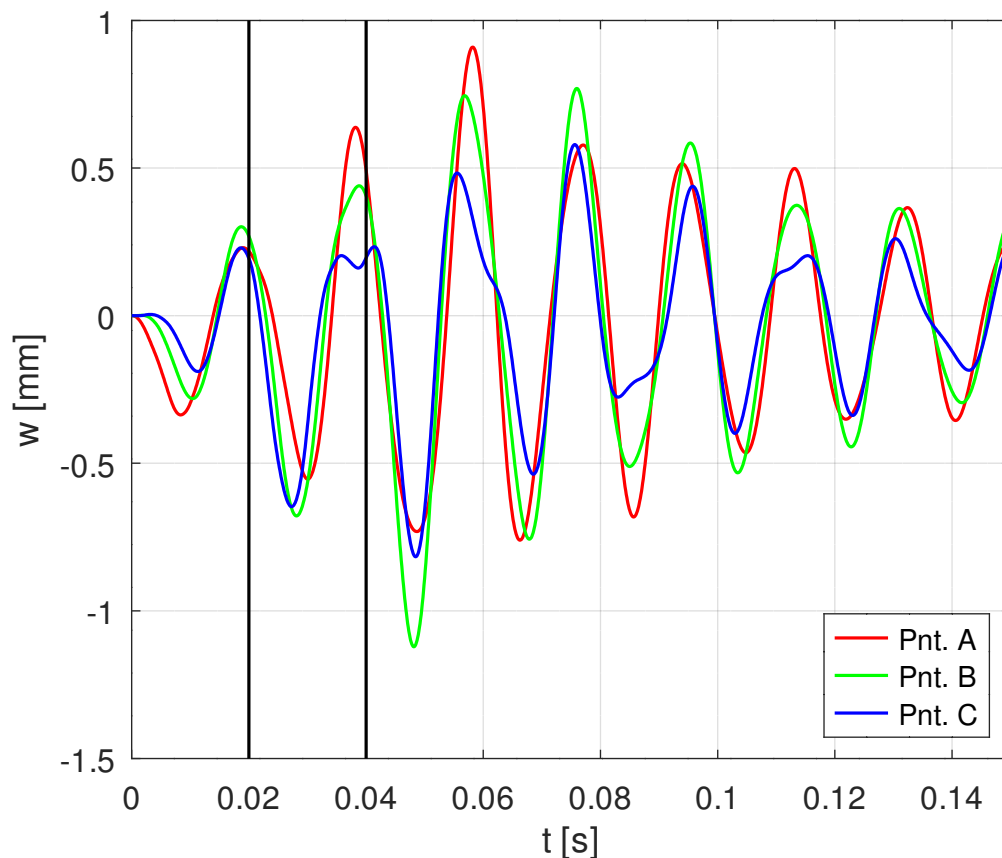


Figure 6.1-6: Plate, Displacements due to Impact Load

```

figure(3, "position", [500, 200, 1000, 750],
      "paperposition", [0, 0, 15, 12]);
plot(t, a);
legend("Pnt. A", "Pnt. B", "Pnt. C");
legend("autoupdate", "off");
grid;
xlim([0, Ts]);
line([delay(2), delay(2)], ylim(), "color", "black");
line([delay(3), delay(3)], ylim(), "color", "black");
xlabel('t [s]'); ylabel('a [g]');

print("impact_modal_a.svg", "-dsvg");

```

Figure 6.1-6 shows the displacements, Figure 6.1-7 the velocities and Figure 6.1-8 the accelerations at points A, B and C. The vertical lines indicate the times at which the impacts occur.

As we want to compare these results with those from a direct transient response analysis, we store them in a csv-file.

```
# Save results
```

```
dlmwrite("impact_modal.csv", [t; w; v; a], "precision", 5);
```

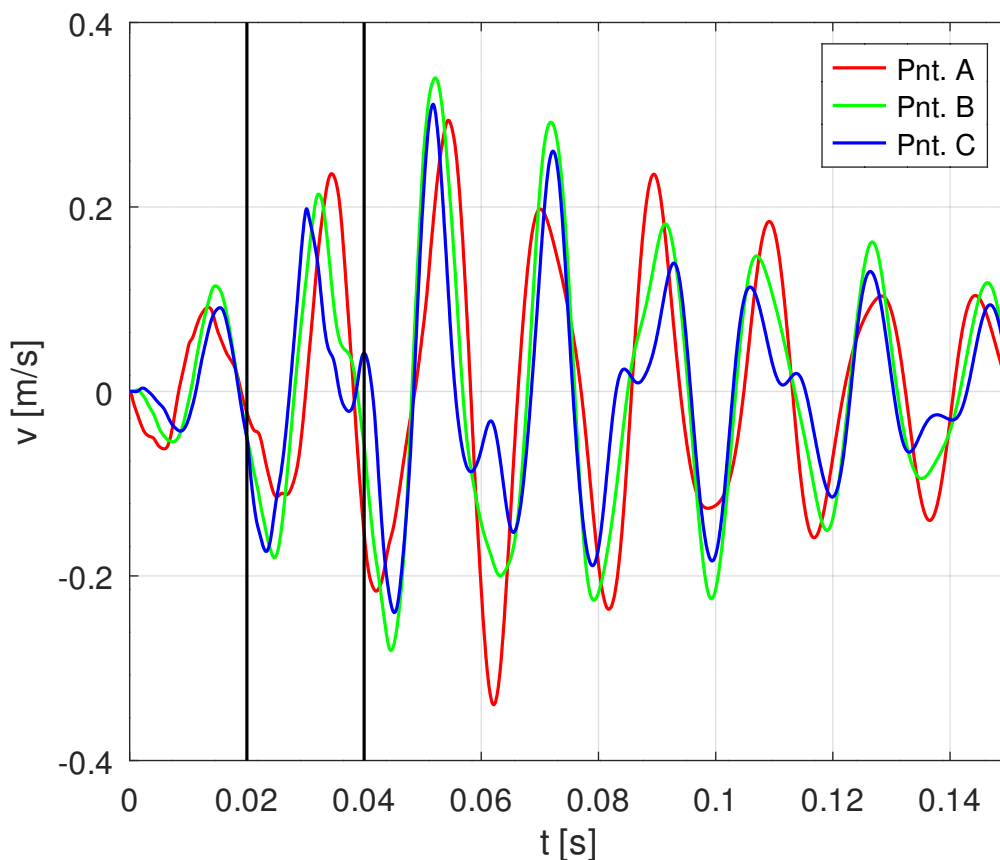


Figure 6.1-7: Plate, Velocities due to Impact Load

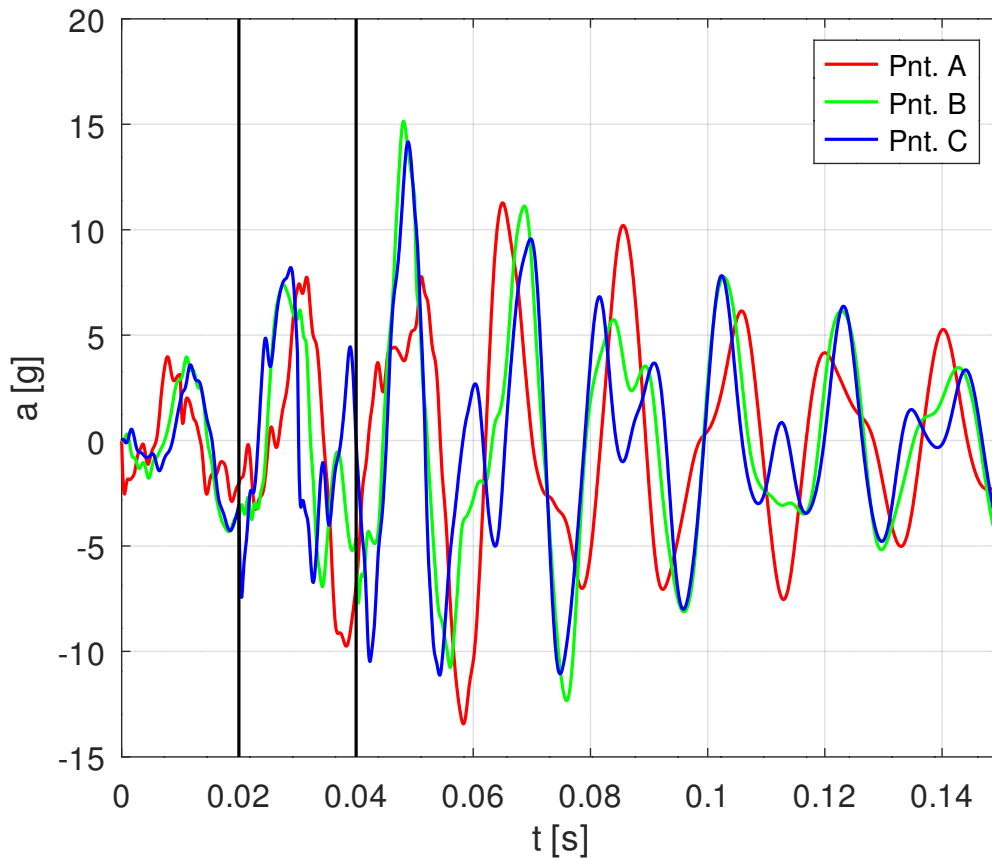


Figure 6.1-8: Plate, Accelerations due to Impact Load

Finally we compute the displacements at all nodal points for selected points in time and write the data to file `impact_modal.dsp`. We do not do this for all time steps because the resulting file would be very large. Subsequently we can use Gmsh to view an animation using the animation buttons as described in Section 3.2.

```
# Perform backtransformation

tb = t(1 : 5 : length(t));
plate = mfs_back(plate, "transresp", "disp", 1, tb);
mfs_export("impact_modal.dsp", "msh", plate,
           "transresp", "disp");

fclose(fid);
```

The enhanced modal transient response analysis is the most commonly used method of transient response analysis. To see how accurate this method is, we repeat the analysis using a direct transient response analysis.

The commands to perform the direct transient response analysis are contained in file `tresp_impact_direct.m`. The only difference is that the method must be defined as additional argument of function `mfs_transresp`.

```
# Perform transient response analysis
```

```
plate = mfs_transresp(plate, [dt, Ts], "load", loads,  
                        "method", "direct");
```

In addition, a direct transient response analysis does not require a back transformation because the displacements have already been computed at all nodal points.

File `compare.m` plots the results at points *A*, *B* and *C* of the direct transient response analysis together with the result of the modal transient response analysis. Figure 6.1-9 shows the displacements, Figure 6.1-10 the velocities and Figure 6.1-11 the accelerations. It can be seen that displacements and velocities are practically identical, while some minor differences can be ob-

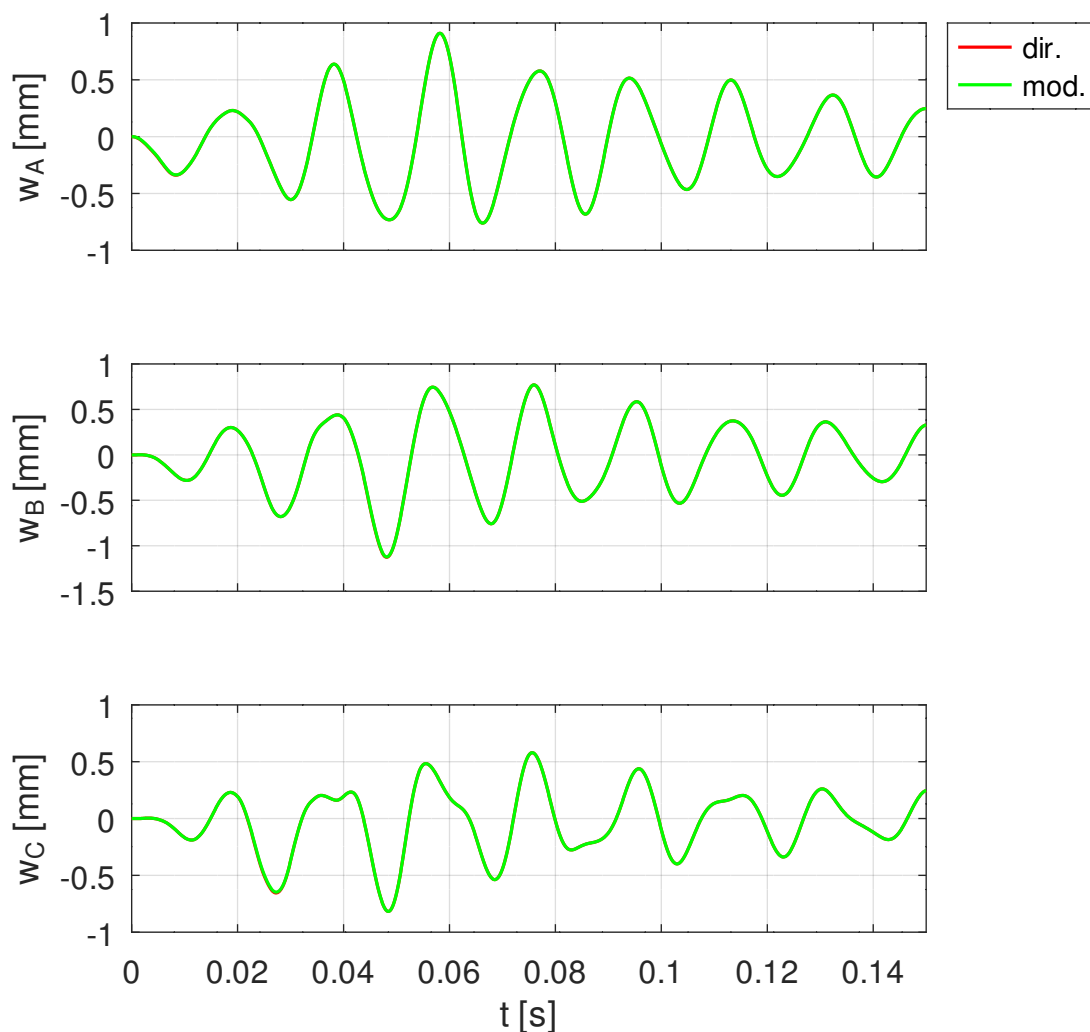


Figure 6.1-9: Plate, Comparison of Displacements due to Impact Load

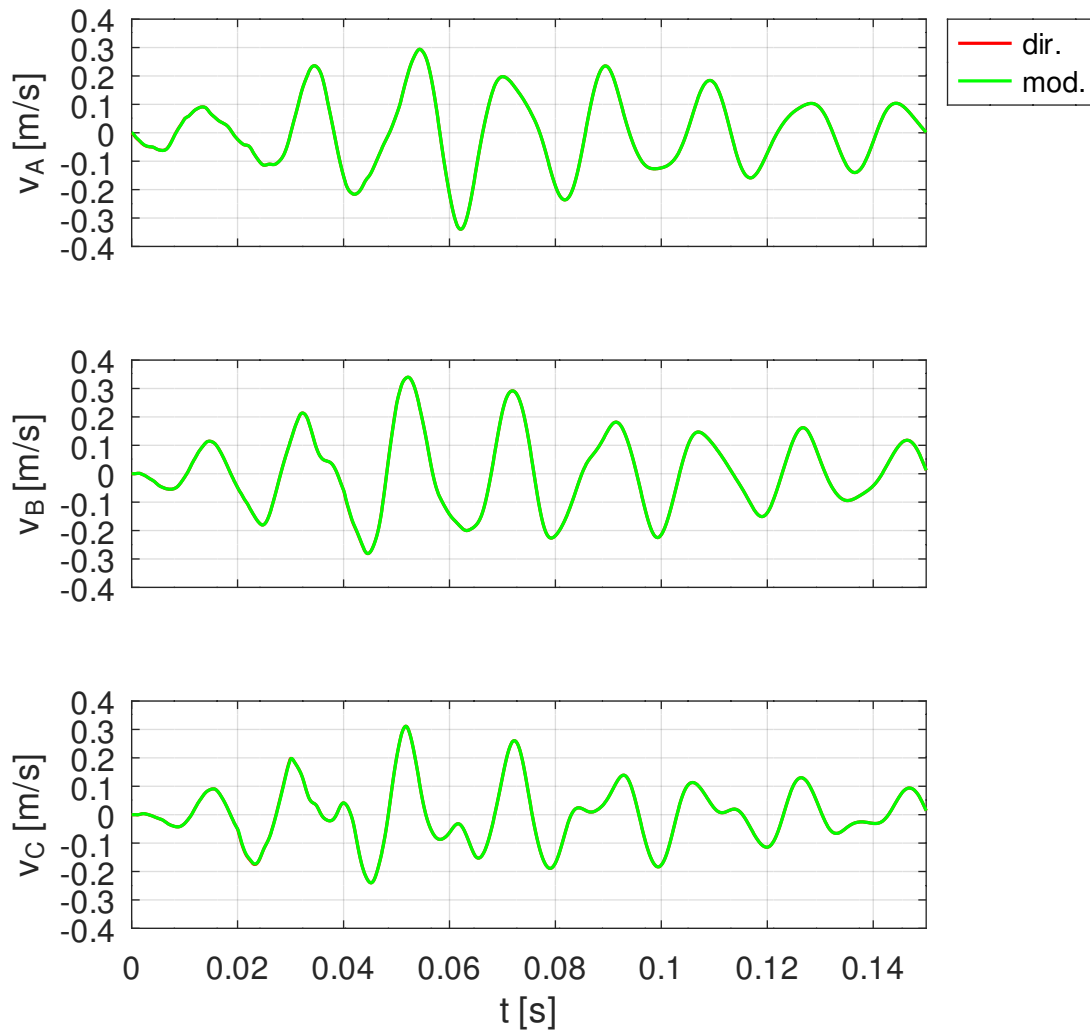


Figure 6.1-10: Plate, Comparison of Velocities due to Impact Load

served in the accelerations.

Next, we examine the situation where the plate is released from static equilibrium. First, we compute the static deformation of load case 1. This deformation defines the initial deformation of the transient response analysis, the initial velocity being zero.

In a modal transient response analysis, the initial conditions are approximated by a linear combination of the normal modes. The quality of this approximation can be assessed using the static strain energies. The modal static strain energies of the first load case sum up to 98 % of the static strain energy. Thus, the 20 computed normal modes should be sufficient to accurately represent the initial deformation.

If the force is removed immediately, i.e. if the transient response analysis starts from the initial deformation without applied force, the accelerations at

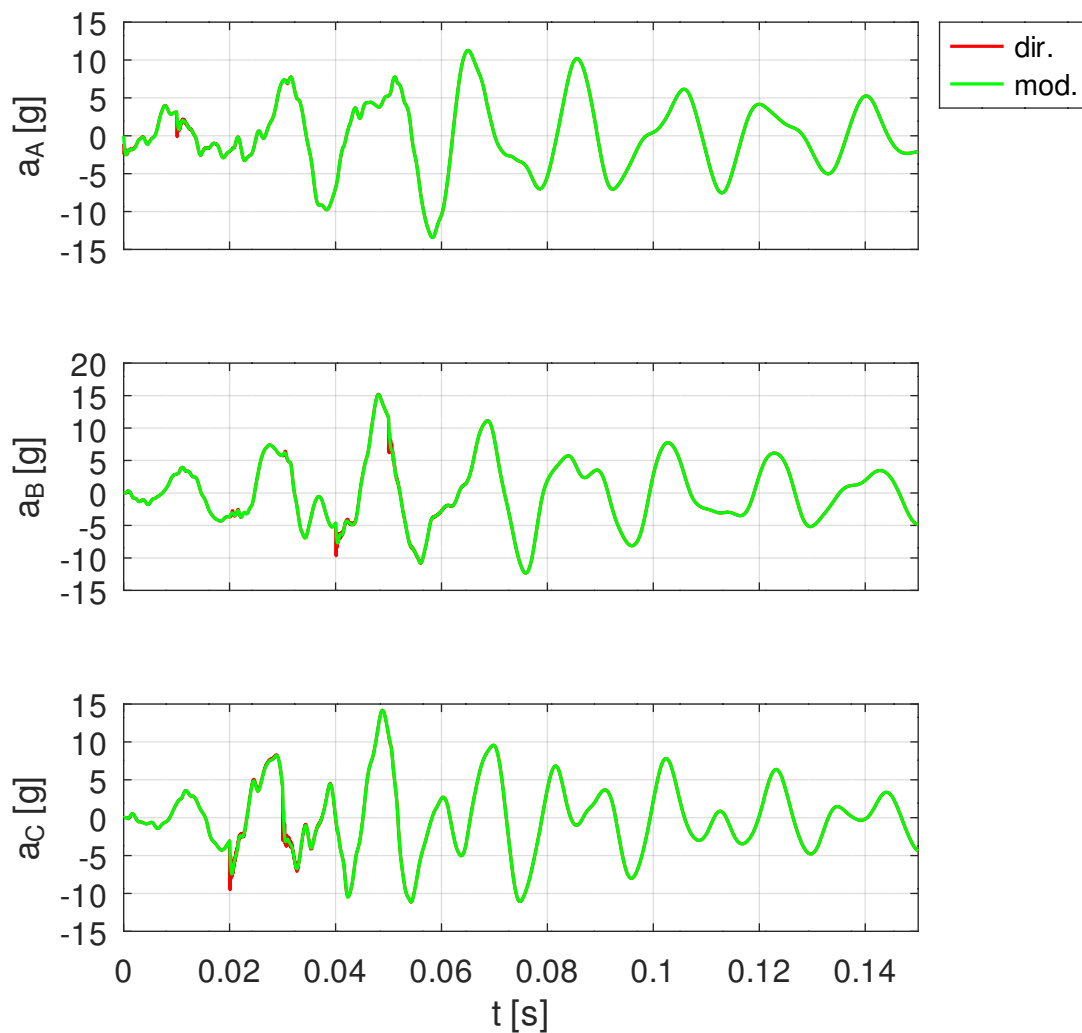


Figure 6.1-11: Plate, Comparison of Accelerations due to Impact Load

the beginning will be unrealistically large. Therefore, we use a decay function to smoothly remove the static load.

The decay function used is

$$\phi(t) = \begin{cases} \cos^2\left(\frac{\pi}{2} \frac{t}{T}\right), & 0 \leq t \leq T \\ 0, & t > T \end{cases}$$

where  $T$  is the duration of the decay. We use a value of 2 ms for this parameter. Decreasing  $T$  leads to an increase in the accelerations occurring at the beginning of the analysis.

Function  $\phi(t)$  is computed by function **decay**:

```
function y = decay(t, params)
```



```

# Input  t(:)      Array with time steps
#         params(1) (1) Duration of decay
#
# Output y(:)      Time history
#
# -----

n = length(t);
y = zeros(1, n);

T = params(1);
ix = find(t <= T);
w = pi / (2 * T);
y(ix) = cos(w * t(ix)).^2;

end

```

File `tresp_decay_modal.m` contains the commands to perform the modal transient response analysis. The file begins with the definition of the duration of the decay, the time step and the simulation time. The values for the time step and the simulation time depend on the resonance frequencies and are therefore the same as for the impact analysis.

```

fid = fopen("decay_modal.res", "wt");

# Define excitation parameters

T      = 2e-3; % Duration of decay

# Define simulation parameters

dt      = 1e-4; % Time step
Ts      = 0.15; % Simulation time

```

Structure `loads` links the first load case to the decay function.

```

# Define excitation

loads = struct("lc", 1, "func", "decay", "params", T);

```

Next, we load the component containing the results of the normal modes analysis and compute the static deformation. The displacements at points *A*, *B* and *C* are stored in array `ws`. We will compare these values with the initial values from the transient response analysis to assess the accuracy of the modal reduction.

```

# Get results from normal modes analysis

if (~ isfile("modes.bin"))
    error("File modes.m must be executed before\n");
end

```

```
load modes.bin

# Compute initial deformation

plate = mfs_statresp(plate);
plate = mfs_newset(plate, "nset", "union",
                  {"Resp_A", "Resp_B", "Resp_C"}, "Resp");
ws     = mfs_getresp(plate, "statresp", "disp", "Resp");
```

Now we perform the transient response analysis. The keyword **"statresp"** indicates that the following argument is the number of a static load case defining the initial deformation. Then the results are plotted and stored for later comparison with the results of a direct transient response analysis.

```
# Perform transient response analysis

plate = mfs_transresp(plate, [dt, Ts], "load", loads,
                      "statresp", 1);

# Plot results

rid = {"Resp_A", 3; "Resp_B", 3; "Resp_C", 3};
```

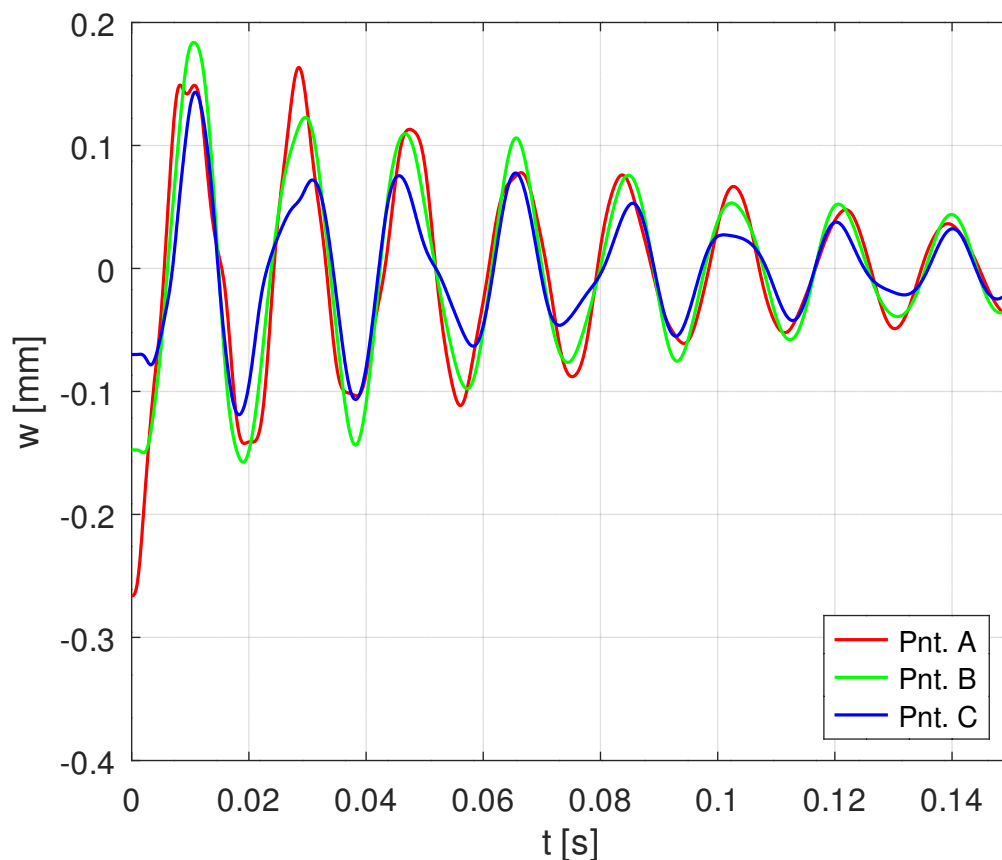


Figure 6.1-12: Plate, Displacements due to Initial Deformation

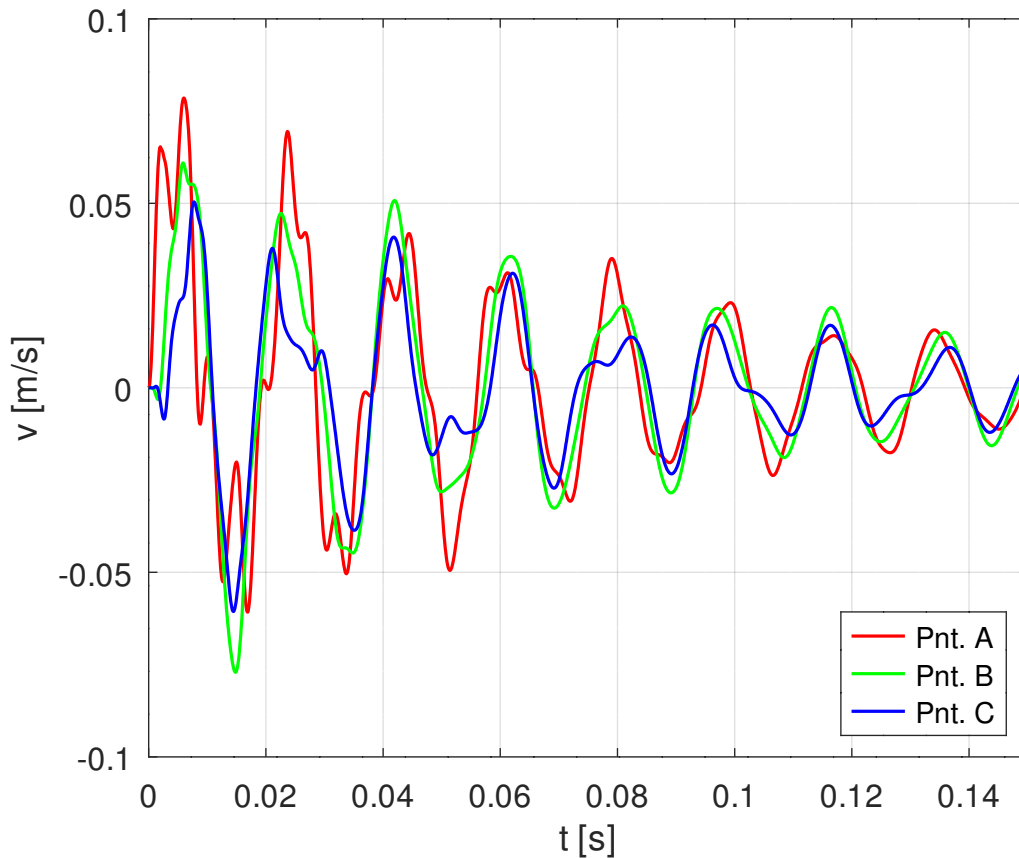


Figure 6.1-13: Plate, Velocities due to Initial Deformation

```

t = mfs_getresp(plate, "transresp", "time");
w = mfs_getresp(plate, "transresp", "disp", rid);
v = mfs_getresp(plate, "transresp", "velo", rid) / 1000;
a = mfs_getresp(plate, "transresp", "acce", rid) / 9810;

figure(1, "position", [100, 200, 1000, 750],
        "paperposition", [0, 0, 15, 12]);
...
# Save results

dlmwrite("decay_modal.csv", [t; w; v; a], "precision", 5);

```

Figures 6.1-12, 6.1-13 and 6.1-14 show the displacements, the velocities and the accelerations at points A, B and C.

Next we compute the differences between the static displacements and the initial displacements at points A, B and C. Based on these values, we can judge how accurately the initial deformation is approximated by the normal modes.

```

# Check accuracy of initial conditions

```

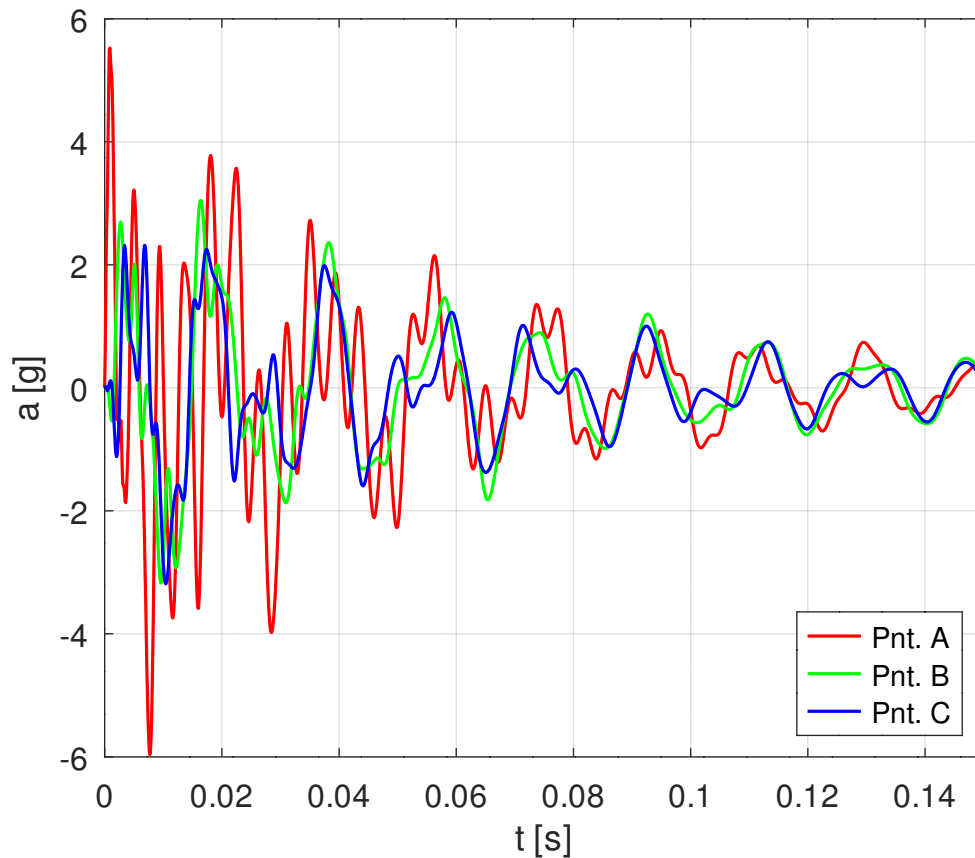


Figure 6.1-14: Plate, Accelerations due to Initial Deformation

```
ew = 100 * (ws(:, 3) - w(:, 1)) ./ ws(:, 3);
fprintf(fid, "    Displacement error:\n");
fprintf(fid, "        Point A: %11.4e %%\n", ew(1));
fprintf(fid, "        Point B: %11.4e %%\n", ew(2));
fprintf(fid, "        Point C: %11.4e %%\n", ew(3));
```

The following results are written to the output file:

```
Displacement error:
Point A: -4.8799e-11 %
Point B: -7.5255e-14 %
Point C: -9.4885e-12 %
```

These values indicate a very good accuracy of the initial displacements.

The last commands compute the displacements at all nodal points for selected points in time and write the data to file `decay_modal.dsp` so that the deformation can be animated in Gmsh.

```
# Compute and export deformed shapes

tb = t(1 : 5 : length(t));
plate = mfs_back(plate, "transresp", "disp", 1, tb);
mfs_export("decay_modal.dsp", "msh", plate,
           "transresp", "disp");
```

```
fclose(fid);
```

Finally, file `tresp_decay_direct.m` contains the commands to perform the corresponding direct transient response analysis. Since there are only minor differences to files `tresp_decay_modal.m` and `tresp_impact_direct.m`, no further comments are necessary. The results of the direct transient response analysis are practically identical to those of the modal transient response analysis. As an example, Figure 6.1-15 shows the comparison of the accelerations.

### Transient Results from a Frequency Response Analysis

The transient response to impulsive loads can also be computed from the fre-

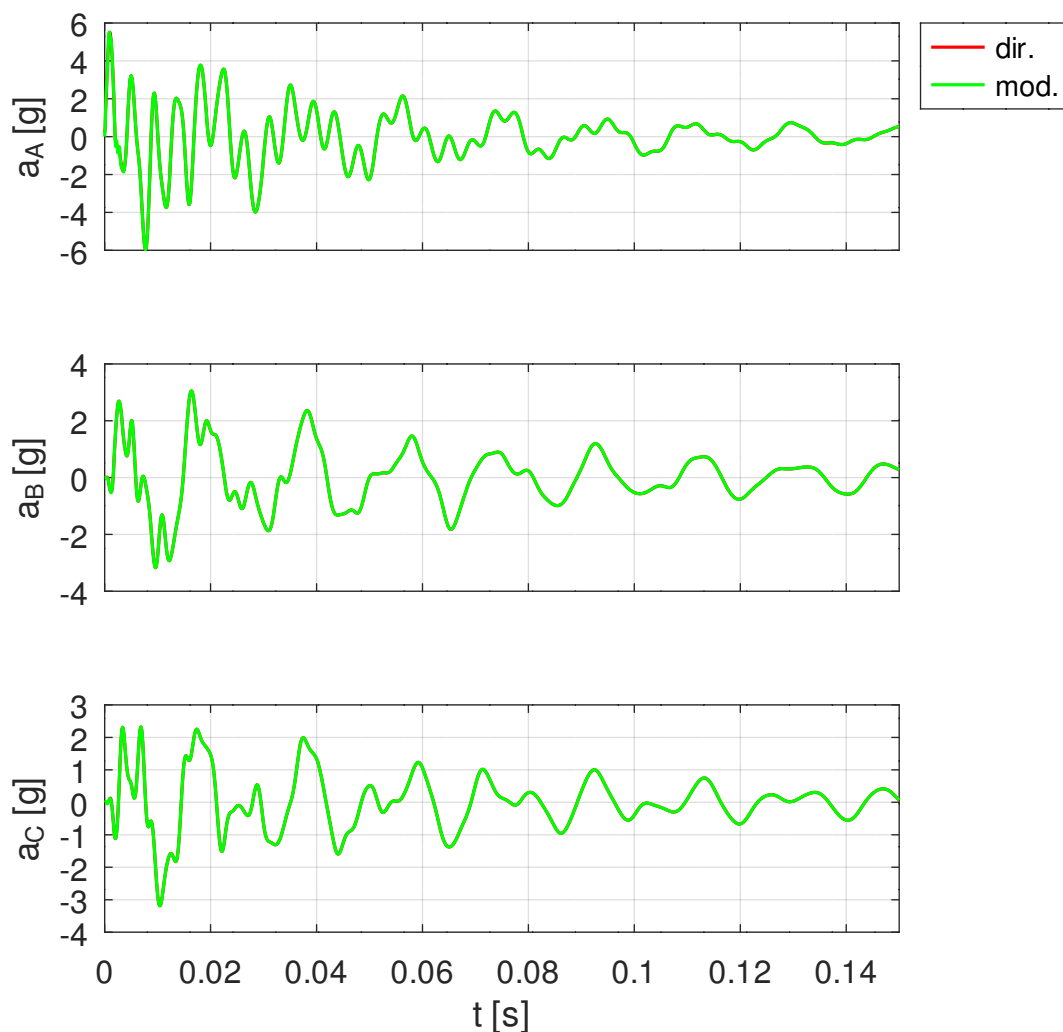


Figure 6.1-15: Plate, Comparison of Accelerations due to Initial Deformation

quency response. First, the transfer functions are computed in a frequency response analysis. Then the transfer functions are multiplied by the Fourier transforms of the loads. The result is the Fourier transforms of the responses. Finally, an inverse Fourier transformation yields the transient responses. This procedure is possible if both the Fourier transforms of the loads and of the responses exist. It is particularly useful in the presence of frequency-dependent matrices. Currently, frequency-dependent matrices in Mefisto are only supported in aeroelasticity.

File `fresp_impact_modal.m` shows how this procedure works when the transfer functions are computed in an enhanced modal frequency response analysis. First, the excitation parameters are defined as for the transient response analysis.

```
# Define excitation parameters

T      =      0.01;    % Duration of pulse
delay = [0, 4, 2] * T; % Delays
```

The simulation parameters are different. We have to define a cut-off frequency and a frequency increment. As found in the analysis of the loads, we define a cut-off frequency of 450 Hz. The values of the Fourier transforms obtained from the Fast Fourier Transformation are given at uniformly distributed frequencies. It is convenient to use the same frequencies in the frequency response analysis so that the frequencies of the Fourier transforms match those of the transfer matrices. We select a frequency increment of 1 Hz. This should be sufficient to compute the transfer functions with satisfactory accuracy. The time step for the transient response obtained from the inverse Fourier transformation as well as the time interval to be plotted is selected as in the transient response analysis.

```
# Define simulation parameters

fc = 450; % Cut-off frequency
df = 1;   % Frequency resolution
dtb = 1e-4; % Time step for backtransformation
Tp  = 0.15; % Time to plot
```

Next, we determine the time step and the duration of the time series for the Fourier transform. The time step is the inverse of the sampling rate. According to the Nyquist theorem, the sampling rate must be twice the cut-off frequency. The frequency increment is the inverse of the duration of the time series.

```
# Compute parameters for Fourier transform

dt = 1 / (2 * fc); % Time step
```

```
Ts = 1 / df;           % Duration of time series
```

Now we load the results from the normal modes analysis and perform the frequency response analysis. There is one analysis for each load case. By default, in a modal frequency response analysis five additional excitation frequencies are added in the halfpower bandwidth of each resonance frequency. In this case, we do not want these frequencies to be added, so we need to specify a number of zero additional frequencies.

```
# Get results from normal modes analysis

if (! isfile("modes.bin"))
    error("File modes.m must be executed before\n");
end

load modes.bin

# Perform frequency response analysis

f = 0 : df : fc; nfreq = length(f);
for lc = 1 : 3
    plate = mfs_freqresp(plate, f, "loadcase", lc, "nband", 0);
endfor
```

Following the frequency response analysis we retrieve the transfer functions for the vertical displacements at points A, B and C. Arrays **HA**, **HB** and **HC** contain the transfer matrices  $[H_A]$ ,  $[H_B]$  and  $[H_C]$  corresponding to the forces at points A, B and C respectively. Each of these matrices has three rows corresponding to the three responses and as many columns as there are excitation frequencies.

```
# Plot transfer functions

rid = {"Resp_A", 3; "Resp_B", 3; "Resp_C", 3};

HA = mfs_getresp(plate, "freqresp", "disp", rid, 1);
HB = mfs_getresp(plate, "freqresp", "disp", rid, 2);
HC = mfs_getresp(plate, "freqresp", "disp", rid, 3);
```

Subsequently, we plot the transfer functions. The resulting picture is shown in Figure 6.1-16. The unit of these transfer functions is length per force.

```
figure(1, "position", [100, 200, 1000, 800],
    "paperposition", [0, 0, 15, 15]);
subplot(3, 1, 1);
semilogy(f, abs(HA));
legend("Pnt. A", "Pnt. B", "Pnt. C",
    "location", "northeastoutside");
grid;
axis("labely");
```

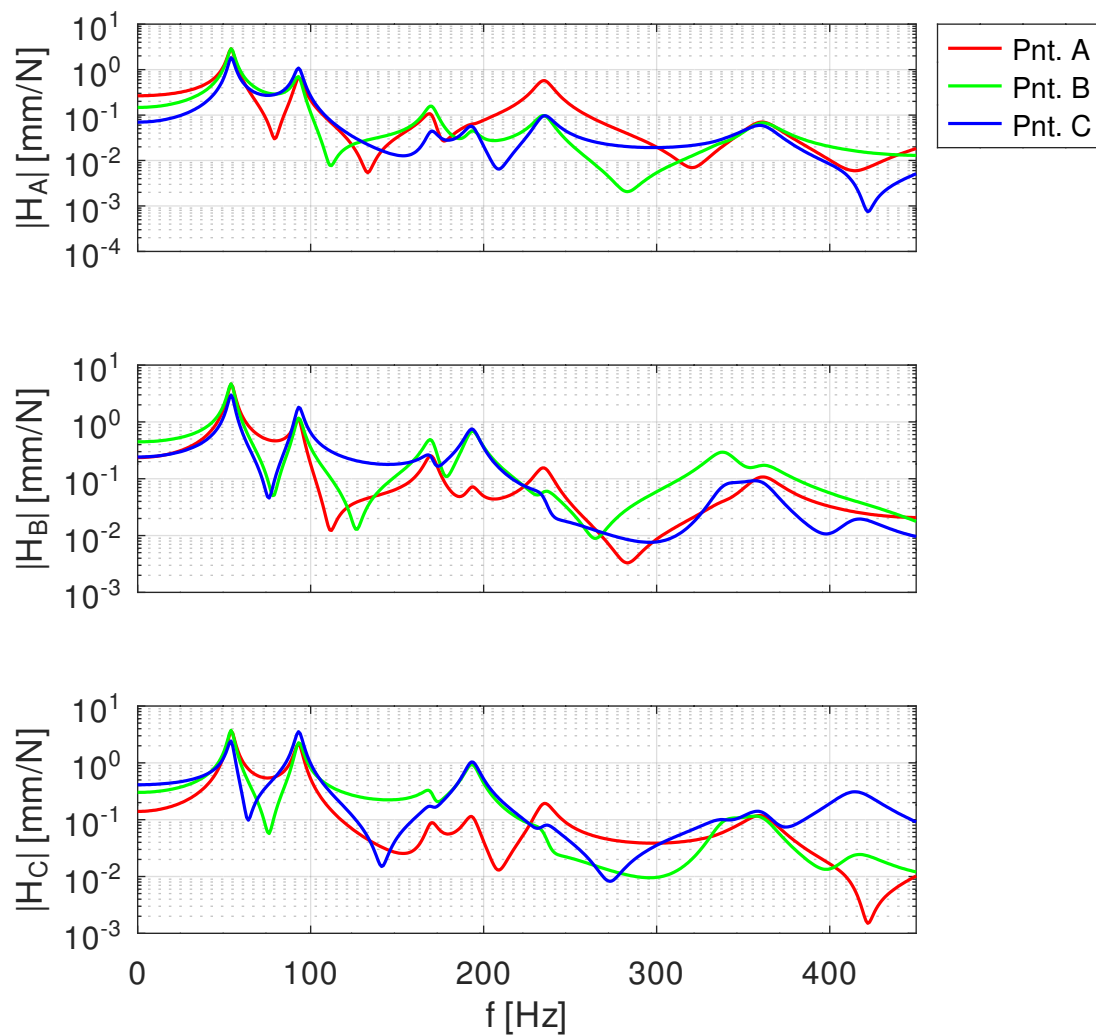


Figure 6.1-16: Plate, Transfer Functions

```

xlim([0, fc]);
ylabel('|H_A| [mm/N]');
subplot(3, 1, 2)
semilogy(f, abs(HB));
grid;
axis("labely");
xlim([0, fc]);
ylabel('|H_B| [mm/N]');
subplot(3, 1, 3)
semilogy(f, abs(HC));
grid;
xlim([0, fc]);
ylabel('|H_C| [mm/N]');
xlabel('f [Hz]');
print("impact_tf.svg", "-dsvg");

```



Next, we compute the time function of the force at point A and its Fourier transform. The first **nfreq** entries in array **L** correspond to the positive frequencies. The time functions of the forces at the other two points are identical to the time function of the force at point A, but shifted by delays  $\Delta t_A$  and  $\Delta t_B$  respectively. Their Fourier transforms are obtained from

$$L_n = L_A e^{-2\pi i f \Delta t_n}, \quad n=B, C,$$

i.e. they have the same magnitude as  $L_A$  but different phase. Hence all three plots must be identical. The unit of the Fourier transform of a load is load per frequency.

```
# Get Fourier transforms of loads

t = 0 : dt : Ts;
l = pulse(t, [T, 0]);
L = fft(l) * dt;

LA = L(1 : nfreq);
LB = LA .* exp(-2 * pi * i * f * delay(2));
LC = LA .* exp(-2 * pi * i * f * delay(3));

figure(2, "position", [300, 200, 1000, 700],
        "paperposition", [0, 0, 15, 12]);
plot(f, abs([LA; LB; LC]));
legend('L_A', 'L_B', 'L_C');
grid;
xlim([0, fc]);
xlabel('f [Hz]'); ylabel('|L| [N/Hz]');
```

Now we can compute the Fourier transforms of the displacements just by multiplying the transfer matrices with the Fourier transforms of the loads and adding the contributions:

$$[W(f)] = \sum_{n \in \{A, B, C\}} [H_n(f)] L_n(f)$$

The rows of matrix  $[W]$  correspond to points A to C and the columns to the frequencies. The velocities  $[V]$  and the accelerations  $[A]$  are obtained by multiplying once and twice respectively with  $2\pi i f$ . The velocities are divided by 1000 and the accelerations by 9.81 to get velocities in m/s/Hz and accelerations in g/Hz. The subsequent commands produce the diagram shown in Figure 6.1-17.

```
# Compute Fourier transform of response

W = HA .* LA + HB .* LB + HC .* LC;
V = 2 * pi * i * f .* W / 1000;
A = 2 * pi * i * f .* V / 9.81;

figure(3, "position", [500, 200, 1000, 800],
```

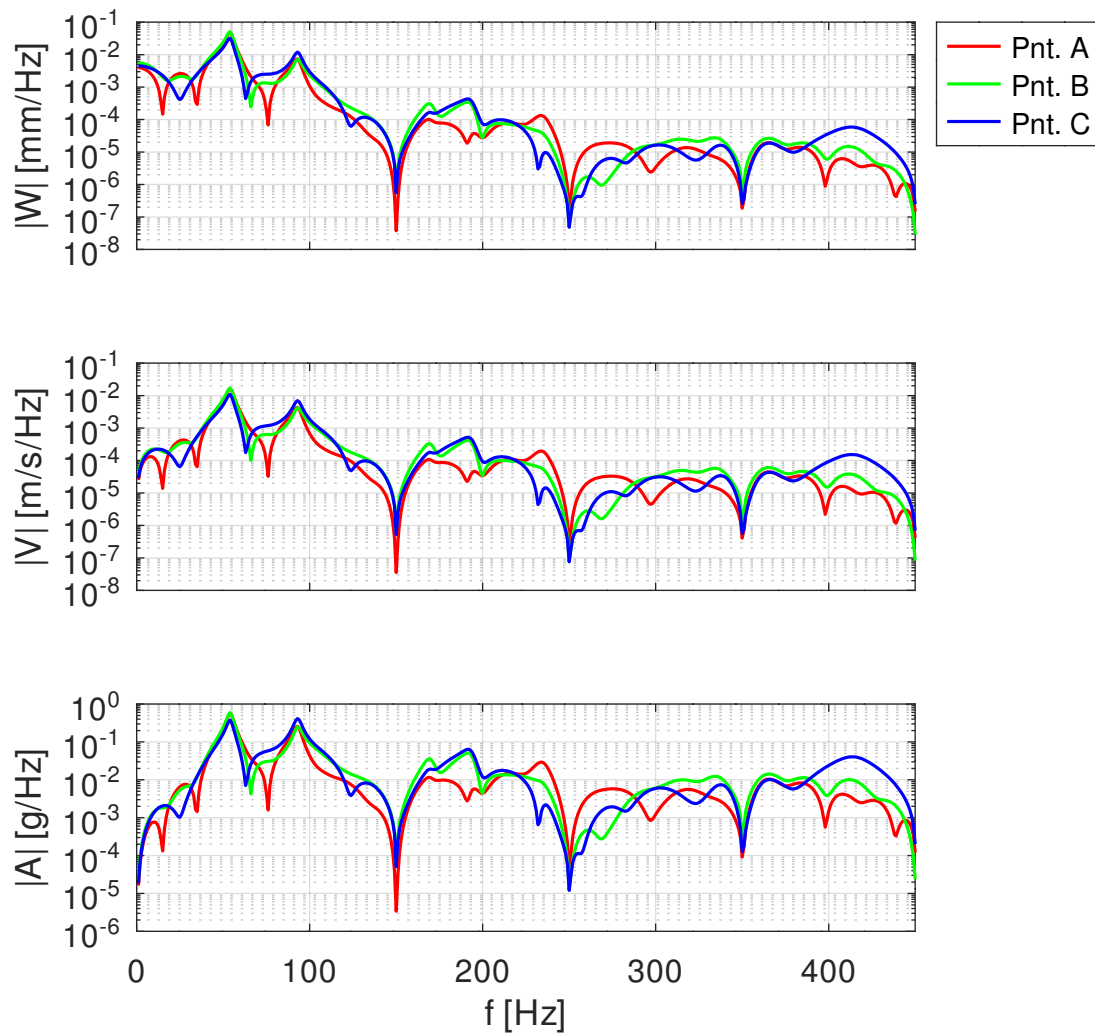


Figure 6.1-17: Plate, Fourier Transforms of Responses

```

        "paperposition", [0, 0, 15, 15]);
subplot(3, 1, 1);
    semilogy(f(2 : end), abs(W(:, 2 : end)));
    legend("Pnt. A", "Pnt. B", "Pnt. C",
        "location", "northeastoutside");
    grid;
    axis("labely");
    xlim([0, fc]);
    ylabel(' |W| [mm/Hz] ');
subplot(3, 1, 2)
    semilogy(f(2 : end), abs(V(:, 2 : end)));
    grid;
    axis("labely");
    xlim([0, fc]);
    ylabel(' |V| [m/s/Hz] ');
subplot(3, 1, 3)
    semilogy(f(2 : end), abs(A(:, 2 : end)));

```

```

    grid;
    xlim([0, fc]);
    ylabel('|A| [g/Hz]');
    xlabel('f [Hz]');
    print("impact_WVA.svg", "-dsvg");

```

The transient response is obtained by an inverse Fourier transformation of the Fourier transforms of the responses. We can use function **mfs\_freq2time** to perform the inverse transformation. The input of this function are the values of the Fourier transform at the positive frequencies, the frequency increment and the time step for the transient response.

```
# Perform transformation to time domain
```

```

[w, t] = mfs_freq2time(W, df, dtb);
[v, t] = mfs_freq2time(V, df, dtb);
[a, t] = mfs_freq2time(A, df, dtb);

```

The remaining commands generate diagrams of the time-dependent displacements, velocities and accelerations and save the results in a csv-file so that we can compare them with the results of a transient response analysis.

```
# Plot time histories
```

```

figure(4, "position", [100, 100, 1000, 750],
        "paperposition", [0, 0, 15, 12]);
plot(t, w);
legend("Pnt. A", "Pnt. B", "Pnt. C", "autoupdate", "off");
grid;
xlim([0, Tp]);
line([delay(2), delay(2)], ylim(), "color", "black");
line([delay(3), delay(3)], ylim(), "color", "black");
xlabel('t [s]'); ylabel('w [mm]');

print("impact_fresp_w.svg", "-dsvg");

figure(5, "position", [300, 100, 1000, 750],
        "paperposition", [0, 0, 15, 12]);
plot(t, v);
legend("Pnt. A", "Pnt. B", "Pnt. C", "autoupdate", "off");
grid;
xlim([0, Tp]);
line([delay(2), delay(2)], ylim(), "color", "black");
line([delay(3), delay(3)], ylim(), "color", "black");
xlabel('t [s]'); ylabel('v [m/s]');

print("impact_fresp_v.svg", "-dsvg");

figure(6, "position", [500, 100, 1000, 750],
        "paperposition", [0, 0, 15, 12]);
plot(t, a);
legend("Pnt. A", "Pnt. B", "Pnt. C", "autoupdate", "off");

```

```

grid;
xlim([0, Tp]);
line([delay(2), delay(2)], ylim(), "color", "black");
line([delay(3), delay(3)], ylim(), "color", "black");
xlabel('t [s]'); ylabel('a [g]');

print("impact_fresp_a.svg", "-dsvg");

# Save results

dlmwrite("impact_fresp.csv", [t; w; v; a], "precision", 5);

```

The results are practically identical to those of a transient response analysis, so the diagrams are not repeated here. As an example, Figure 6.1-18 compares the accelerations. Only some minor differences can be observed.

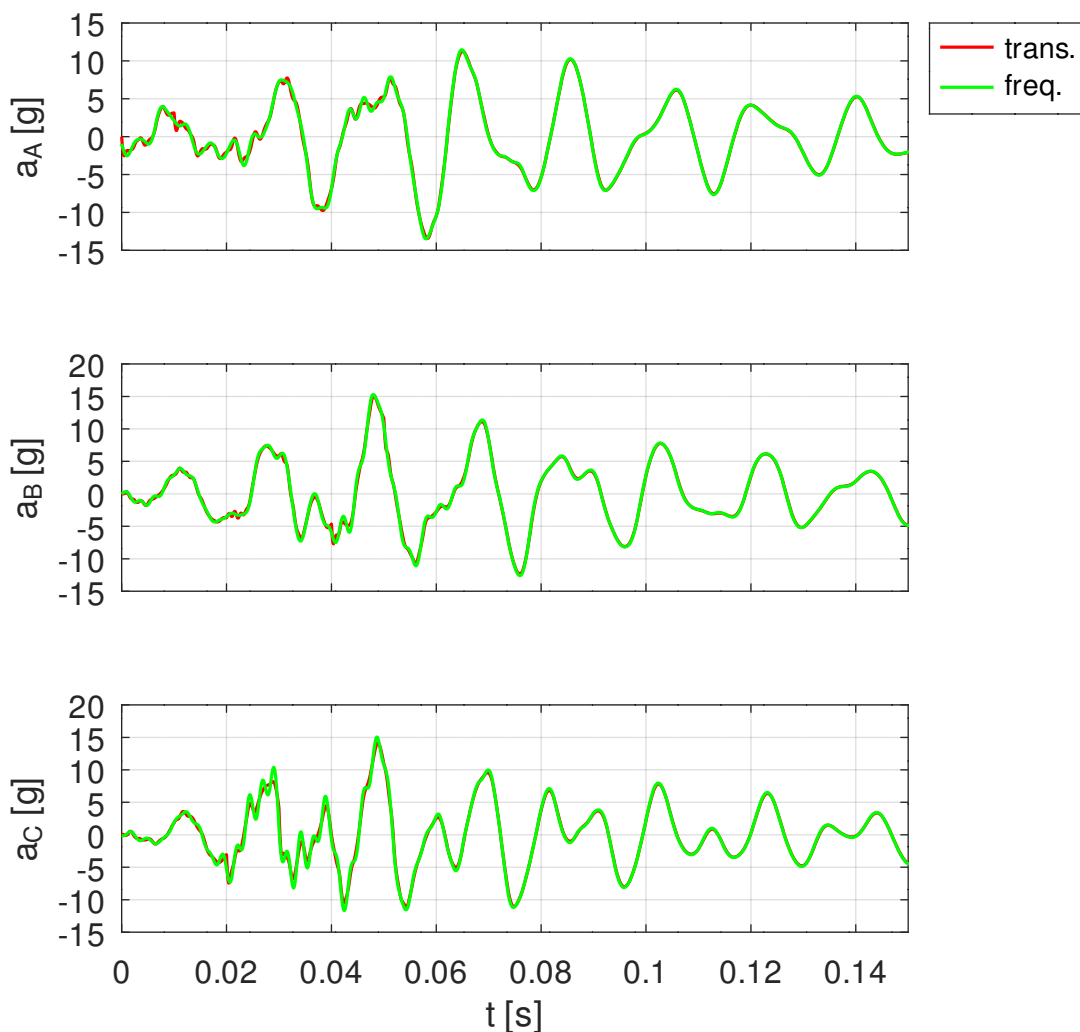


Figure 6.1-18: Plate, Accelerations from Transient and Frequency Response Analysis

## 6.2 Box Beam

### Summary

Directory:	exa/solid/transresp/boxbeam
Objectives:	<ul style="list-style-type: none"> <li>• learn how to define node and element sets using coordinates</li> <li>• learn how to find the cut-off frequency of a transient load</li> <li>• learn how to define a prescribed motion</li> <li>• learn how to define initial conditions</li> <li>• learn how to define modal damping ratios</li> <li>• learn how to run a modal transient response analysis</li> <li>• learn how to retrieve transient responses</li> <li>• learn how to back-transform results from a transient response analysis</li> <li>• learn how to compute transient results from the results of a frequency response analysis</li> </ul>
Dimension:	3
Elements:	<b>s4</b>
Loads:	prescribed motion
Functions:	<b>mfs_import, mfs_new, mfs_stiff, mfs_mass, mfs_massproperties, mfs_freevib, mfs_print, mfs_export, mfs_reductionerror, mfs_newset, mfs_getset, mfs_transresp, mfs_getresp, mfs_back, mfs_results, mfs_freqresp, mfs_freq2time</b>

### Problem Description

In this example, we examine the transient response of the cantilever beam with a thin-walled box cross section shown in Figure 6.2-1. This is the same cantilever beam that was already studied in Example 4.2. However, the damping of the beam is now described by modal damping ratios and the excitation is a prescribed ramp-like vertical motion of the restrained left end.

The vertical displacements at the left end are given by

$$u_z(t) = u_{z0} \phi(t)$$

where

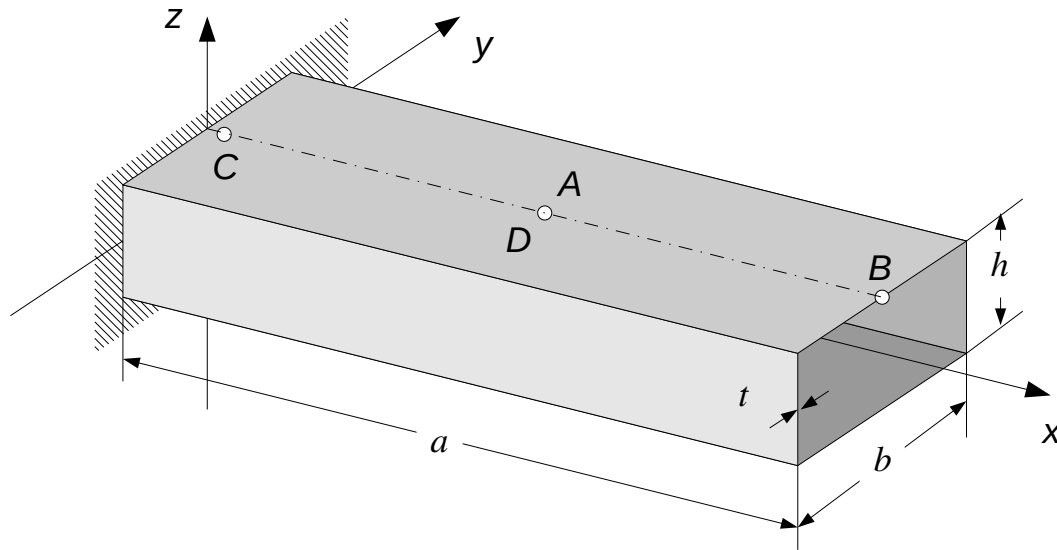


Figure 6.2-1: Box Beam Structure

$$\phi(t) = \begin{cases} \frac{t}{T} - \frac{1}{2\pi} \sin\left(2\pi \frac{t}{T}\right), & 0 \leq t \leq T \\ 1, & t > T \end{cases}.$$

The results of interest are

1. the vertical displacements, velocities and accelerations at points A and B and
2. the normal stresses in x-direction at the upper side of the upper shell at points C and D.

Data:  $a = 1000$  mm,  $b = 100$  mm,  $h = 50$  mm,  $t = 3$  mm, Young's modulus  $E = 210$  GPa, Poisson's ratio  $\nu = 0,3$ , mass density  $\rho = 7850$  kg/m<sup>3</sup>, modal damping ratios of 5 % for the first five normal modes and of 2 % for all remaining normal modes, displacement amplitude  $u_{z0} = 10$  mm, duration of motion  $T = 0.02$  s

### Analysis of the Load

As in Example 6.1 we start with a Fourier transformation of the load.

In case of a base motion excitation, we must provide a function that computes the time dependence of the displacements, the velocities and the accelerations, i.e. we also need the derivatives

$$\dot{\phi}(t) = \begin{cases} \frac{1}{T} \left( 1 - \cos\left(2\pi \frac{t}{T}\right) \right), & 0 \leq t \leq T \\ 0, & t > T \end{cases}$$

and

$$\ddot{\phi}(t) = \begin{cases} \frac{2\pi}{T^2} \sin\left(2\pi \frac{t}{T}\right), & 0 \leq t \leq T \\ 0, & t > T \end{cases}.$$

It can be seen that  $\dot{\phi}(0)=\dot{\phi}(T)=0$  and  $\ddot{\phi}(0)=\ddot{\phi}(T)=0$ , i.e. the beam is at rest at the beginning and end of the motion.

Function  $\phi(t)$  and its derivatives are computed by function **motion**:

```
function y = motion(t, T)

# Input    t(:)      Array with time steps
#          T         Duration of motion
#
# Output   y(1, :)   Displacements
#          y(2, :)   Velocities
#          y(3, :)   Accelerations
#
# Structure moves to upper position.
#
# -----

n = length(t);
y = zeros(3, n);

ix = find(t <= T); nx = ix(end);
tx = t(ix) / T;
s = sin(2 * pi * tx); c = cos(2 * pi * tx);

y(1, ix) = tx - s / (2 * pi); y(1, nx + 1 : n) = 1;
y(2, ix) = (1 - c) / T;
y(3, ix) = 2 * pi * s / T^2;

endfunction
```

The Fourier transform of the displacements does not exist because function  $\phi(t)$  does not go to zero if  $t$  goes to infinity. However, the Fourier transforms of the velocities and accelerations do exist. Therefore we can look at the Fourier transform of the accelerations to determine the cut-off frequency.

File **excitation.m** contains the commands for analysing function  $\phi(t)$  and its derivatives. The **signal** package is needed because it contains function **resample** that we will use to check the cut-off frequency. Then, we define the duration  $T$  of the pulse, the length of the time signal, the time step and the cut-off frequency. The length of the time signal is defined here in multiples of

the duration of the pulse. With the given values, the frequency resolution will be

$$\Delta f = \frac{1}{nT} = \frac{1}{50 \cdot 0.02 \text{ s}} = 1 \text{ Hz} .$$

The time step determines the sampling rate  $f_s$  and the Nyquist frequency  $f_a$ . With the given data, we get

$$f_a = \frac{1}{2} f_s = \frac{1}{2 \Delta t} = \frac{40}{2 \cdot 0.02 \text{ s}} = 1000 \text{ Hz} .$$

The values of the Fourier transforms should be negligible at frequencies above the Nyquist frequency.

Finally, we specify a cut-off frequency of 400 Hz. We will check how well the inverse Fourier transform of Fourier transform truncated at this cut-off frequency matches the original function.

```
pkg load signal

# Data

T =      0.02;    % Duration of motion
n =       50;    % Length of time series (multiples of T)
dt = T / 40;    % Time step
fc =     400;    % Cut-off frequency
```

Next, we evaluate function  $\phi(t)$  and its first and second derivatives and plot the resulting curves. The first derivative is multiplied by  $T$  and the second derivative by  $T^2$  so that all functions are dimensionless. The resulting diagram is shown in Figure 6.2-2.

```
# Function values

t = 0 : dt : T;
y = motion(t, T);

y(2, :) *= T;
y(3, :) *= T^2;

figure(1, "position", [100, 100, 750, 750],
        "paperposition", [0, 0, 15, 15]);
subplot(3, 1, 1)
plot(t, y(1, :));
grid;
axis("labely");
ylabel('\phi');
subplot(3, 1, 2)
plot(t, y(2, :));
grid;
axis("labely");
ylabel('T d\phi/dt');
```



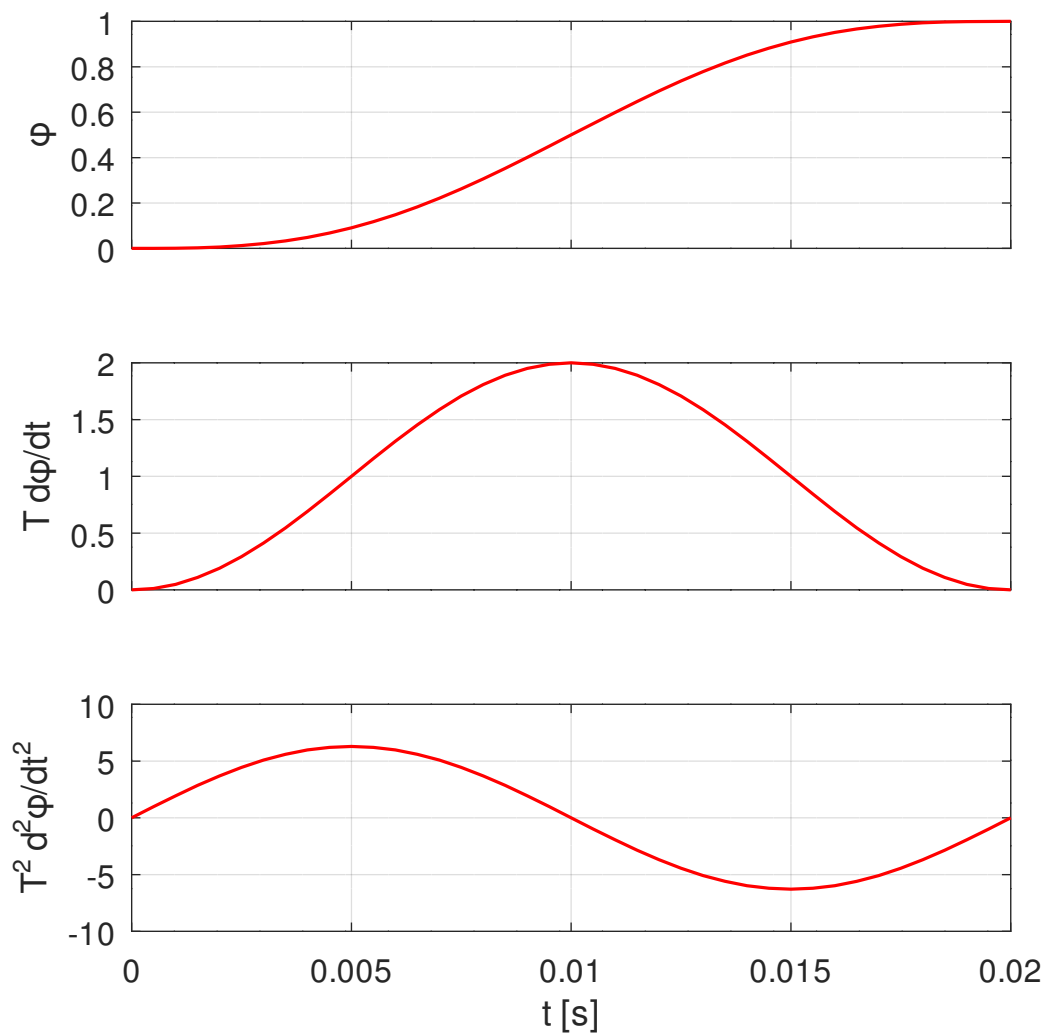


Figure 6.2-2: Box Beam, Excitation Function and Derivatives

```
subplot(3, 1, 3)
plot(t, y(3, :));
grid;
ylabel('T^2 d^2\phi/dt^2');
xlabel('t [s]');
print("motion.svg", "-dsvg");
```

Next, we compute the Fourier transform of the second derivative. The second input argument of function `fft`, variable `N`, defines the length of the time series. The input array `y` will be filled up with zeros so that its length is `N`.

```
# Fourier transform of acceleration

fs = 1 / dt; fa = 0.5 * fs;
N = n * (length(t) - 1);
Ts = n * T;
A = fft(y(3, :), N) * dt;
```

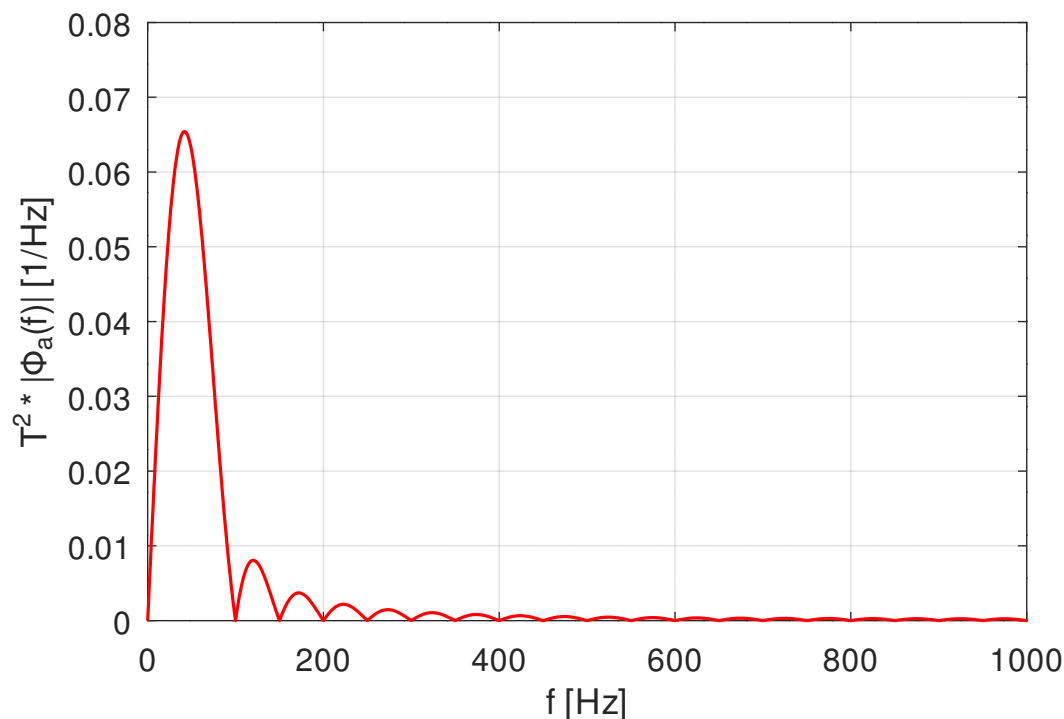


Figure 6.2-3: Box Beam, Fourier Transform of Acceleration

```
f = (0 : N - 1) / Ts;

figure(2, "position", [400, 100, 750, 500],
       "paperposition", [0, 0, 15, 10]);
plot(f, abs(A));
grid;
xlim([0, fa]);
xlabel('f [Hz]'); ylabel('T^2 * |\Phi_a| [1/Hz]');
print("FT.svg", "-dsvg");
```

Figure 6.2-3 shows that the values of the Fourier transform of the acceleration are indeed negligible above 1000 Hz and very small above 400 Hz so that a cut-off frequency of 400 Hz is reasonable. This is confirmed by Figure 6.2-4, which compares the original function with the filtered function obtained from function `resample`. It can be seen that both functions are practically identical.

```
# Test of cut-off frequency

yc = resample(y(3, :), fc, fa);
tc = (0 : length(yc)-1) * dt * fa / fc;

figure(3, "position", [600, 100, 750, 500],
       "paperposition", [0, 0, 15, 10]);
plot(t, y(3, :), tc, yc);
legend("orig.", "filtered");
grid;
xlabel('t [s]'); ylabel('T^2 * d^2\phi/dt^2');
```

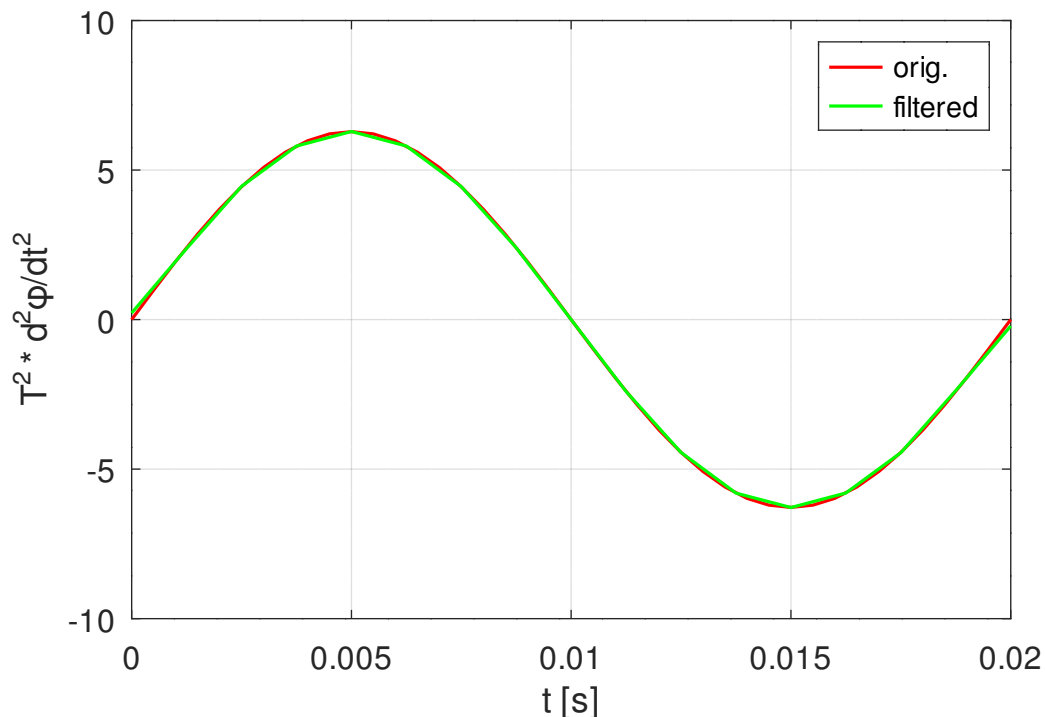


Figure 6.2-4: Box Beam, Original and Filtered Function

```
print("cutoff.svg", "-dsvg");
```

### Model Definition and Normal Modes Analysis

The model is defined in Gmsh. The following physical groups are defined:

- **Shell**: shell elements
- **Constraints**: used to apply the constraints at the left end
- **Motion**: used to apply the prescribed acceleration at the left end

The commands to define the geometry and the mesh are contained in file `beam.geo`. They are identical to those described in Section 4.2, except that physical group **Load** is missing.

File `modes.m` contains the model definition and the commands to perform the normal modes analysis. This file is almost identical to file `modes.m` described in Example 4.2, so only the differences need to be explained.

In the data, we define a displacement amplitude instead of an acceleration and a force amplitude. The number of normal modes to compute is now 10, and the cut-off frequency is 400 Hz.

```
...
# Data (N, mm):
```

```

E    = 210000; % Young's modulus
ny   = 0.3;   % Poisson's ratio
rho  = 7.85E-9; % Mass density
t    = 3;    % Thickness
uz   = 10;   % Displacement amplitude

# Simulation paramters

nofmod = 10; % Number of normal modes
fmax   = 400; % Cut-off frequency

# Translation data

...

```

In a transient response analysis, the load pattern defining a prescribed motion is always a displacement pattern. We assign a vertical prescribed displacement of amplitude **uz** to the physical group **Motion**.

```

BEAM.Motion = struct("type", "loads",
                    "name", "disp",
                    "data", [0, 0, uz]);

```

The modal damping ratios are defined in a cell array. The first cell of the array is itself a cell array assigning a damping ratio of 0.05 to the first five normal modes. The second cell assigns a damping ratio of 0.02 to all remaining normal modes. The cell array must be enclosed by additional curly brackets to prevent the creation of a structure array.

```

BEAM.damping = struct("type", "ratios",
                     "data", {{{5, 0.05}, 0.02}});

# Import model

model = mfs_import(fid, "beam.msh", "msh", BEAM);

...

# Estimate reduction error

mfs_reductionerror(fid, beam, fmax);

```

Unfortunately, there are no geometrical entities corresponding to points *A*, *B*, *C* and *D*. Therefore we cannot use physical groups to define corresponding sets. Instead we use coordinates of points that are close to the points *A*, *B*, *C* and *D*. These coordinates are written to file **rpoints.m** by Gmsh. We then use function **mfs\_newset** to create the required node and element sets. With function **mfs\_getset** we obtain the identifiers of the items in the sets which

we write to the output file so that we can check them.

```
# Sets for response

rpoints

beam = mfs_newset(beam, "nset", "near", pntA, "Point_A");
beam = mfs_newset(beam, "nset", "near", pntB, "Point_B");
beam = mfs_newset(beam, "eset", "near", pntC, "Elem_C");
beam = mfs_newset(beam, "eset", "near", pntD, "Elem_D");

idA = mfs_getset(beam, "nset", "Point_A");
idB = mfs_getset(beam, "nset", "Point_B");
idC = mfs_getset(beam, "eset", "Elem_C");
idD = mfs_getset(beam, "eset", "Elem_D");

fprintf(fid, "  Point_A = %4.0d, Point_B = %4.0d\n",
        idA, idB);
fprintf(fid, "  Elem_C  = %4.0d, Elem_D  = %4.0d\n",
        idC, idD);

# Save results

save -binary modes.bin beam

fclose(fid);
```

The results of the normal mode analysis are the same as in Section 4.2, except that only the first ten normal modes are computed. The output from the error estimation is:

```
Modal strain energies of component "beam"

Loadcase 1:

mode   frequency      En/ES      Sum      1 - Sum
1      74.19 Hz      9.89058e-01  0.989058  1.09421e-02
2      108.73 Hz     4.39319e-27  0.989058  1.09421e-02
3      418.84 Hz     9.98749e-03  0.999045  9.54587e-04
4      556.14 Hz     6.94655e-29  0.999045  9.54587e-04
5      615.79 Hz     3.09560e-30  0.999045  9.54587e-04
6      718.35 Hz     1.17321e-28  0.999045  9.54587e-04
7      837.86 Hz     1.45894e-30  0.999045  9.54587e-04
8      922.21 Hz     6.63965e-04  0.999709  2.90622e-04
9     1017.04 Hz     4.16652e-29  0.999709  2.90622e-04
10    1029.92 Hz     2.57090e-30  0.999709  2.90622e-04

Upper bound on relative strain energy error (fmax = 400.00 Hz)
with static correction: 5.6209e-05
without static correction: 2.0152e-04
```

It can be seen that with static correction, ten normal modes are sufficient to obtain a relative strain energy error of less than 0.006 %. The most important normal modes are the first and the third one. Figures 4.2-3 and 4.2-4 show that, as expected, these are the first and the second vertical bending mode.

## Transient Response Analysis

In a first analysis, we apply the ramp-like prescribed motion to the left end of the beam and compute the response for a time of about  $5T_1$ , where  $T_1=1/f_1$  is the period of the first vertical bending mode. The commands are contained in file `tresp.m`.

First, we specify the duration of the prescribed motion, the number of time intervals of the shortest period and the simulation time in multiples of the longest period.

```
# Define excitation parameters

T      = 0.02;    % Duration of prescribed motion

# Define simulation parameters

ntip = 10;    % Number of time intervals per period
ntl  = 5;    % Simulation time as multiple of longest period
```

Next, we load the results from the normal modes analysis and define the degrees of freedom for displacement, velocity and acceleration output and the elements for stress output. Subsequently, we load the results from the normal modes analysis.

```
# Get results from normal modes analysis

if (~ isfile("modes.bin"))
    error("File modes.m must be executed before\n");
end

load modes.bin

# Degrees of freedom for output

rid = {"Point_A", 3; "Point_B", 3};
```

In this example, we compute the simulation time and the time step directly from the resonance frequencies. The simulation time **Ts** is computed based on the first resonance frequency and rounded to hundredths of a second. The time step **dt** is computed using the highest resonance frequency, rounded to multiples of thousands.

```
# Get simulation time and time step from natural frequencies

f = mfs_getresp(beam, "modes", "freq");
Ts = ceil(100 * ntl / f(1)) / 100;
fm = 1000 * ceil(f(end) / 1000);
dt = 1 / (ntip * fm);
```

```
fprintf(fid, "\nSimulation parameters:\n\n");
fprintf(fid, "  Simulation time = %12.6e s \n", Ts);
fprintf(fid, "  Time step      = %12.6e s\n\n", dt);
```

The computed values are written to the output file.

Simulation parameters:

```
Simulation time = 7.000000e-02 s
Time step      = 5.000000e-05 s
```

Next, we define the enforced motion and perform an enhanced modal transient response analysis. Structure **motdef** assigns function **motion** to load case 1 which defines the displacement pattern.

```
# Define enforced motion

motdef = struct("lc", 1, "func", "motion", "params", T);

# Perform modal transient response analysis

beam = mfs_transresp(beam, [dt, Ts], "load", motdef);
```

Now the vertical displacements, velocities and accelerations at points A and B can be plotted. Figure 6.2-5 shows the resulting diagrams. It can be seen that the response is dominated by the first vertical bending mode. Subsequently, the results are saved in file **tresp.csv**.

```
# Plot results at points A and B

t = mfs_getresp(beam, "transresp", "time");
u = mfs_getresp(beam, "transresp", "disp", rid);
v = mfs_getresp(beam, "transresp", "velo", rid) / 1000;
a = mfs_getresp(beam, "transresp", "acce", rid) / 9810;

figure(1, "position", [100, 200, 1000, 800],
       "paperposition", [0, 0, 15, 15]);
subplot(3, 1, 1)
plot(t, u);
legend("Point A", "Point B",
       "location", "southeast");
grid;
axis("labely");
xlim([0, Ts]);
ylabel('u [mm]');
subplot(3, 1, 2)
plot(t, v);
grid;
axis("labely");
xlim([0, Ts]);
ylabel('v [m/s]');
subplot(3, 1, 3)
```

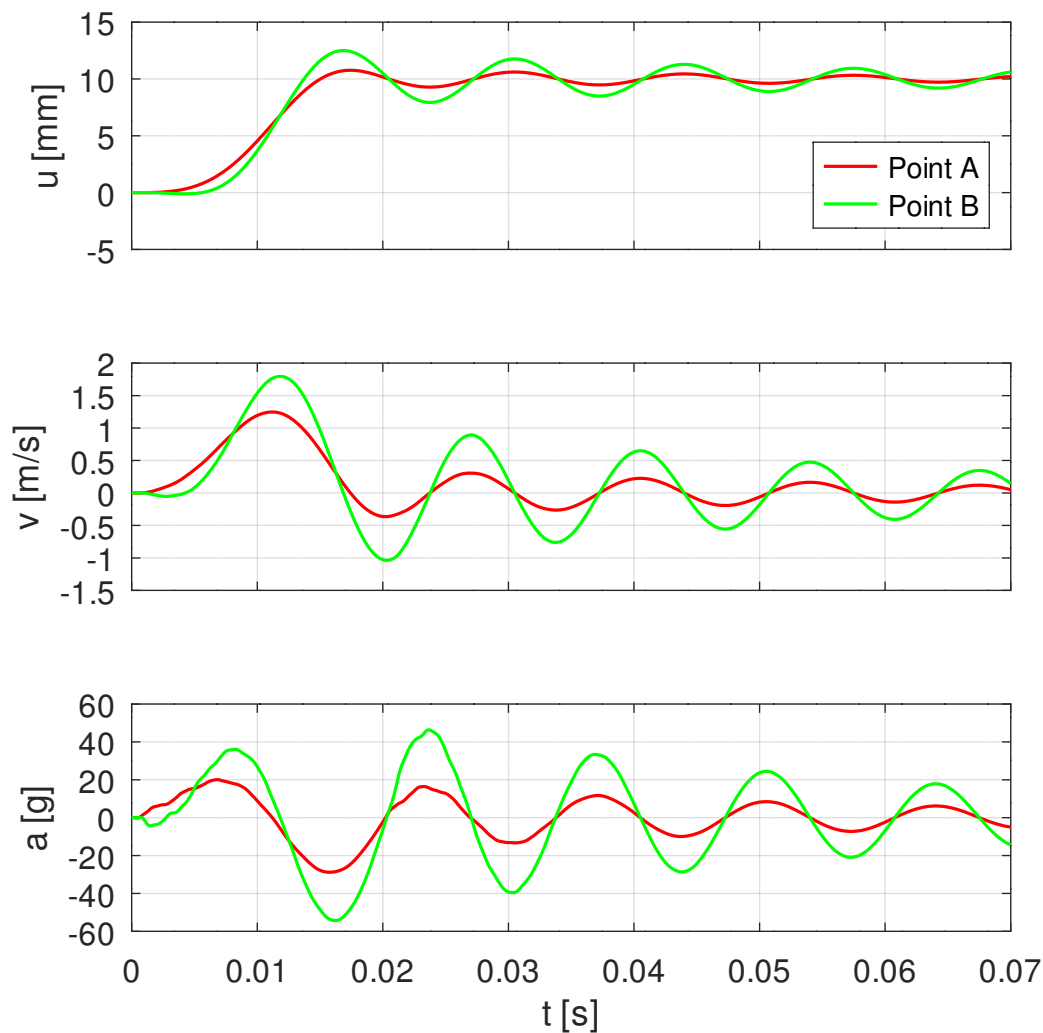


Figure 6.2-5: Box Beam, Displacements, Velocities and Accelerations

```

plot(t, a);
grid;
xlim([0, Ts]);
ylabel('a [g]'); xlabel('t [s]');
print("tresp.svg", "-dsvg");

# Save results

dlmwrite("tresp.csv", [t; u; v; a], "precision", 5);

```

Next, the stresses in elements *C* and *D* are plotted. Function **mfs\_getresp** returns a cell array. Each cell contains a structure with the element results of one element. The normal stresses  $\sigma_x$  at the top of the element are contained in field **sigxu**. Figure 6.2-6 shows the resulting diagram.

```

# Plot stresses in elements C and D

```



```

sig = mfs_getresp(beam, "transresp", "stress",
                  {"Elem_C", "Elem_D"});
sigx = [sig{1}.sigxu; sig{2}.sigxu];

figure(2, "position", [300, 200, 1000, 750],
       "paperposition", [0, 0, 15, 10]);
plot(t, sigx);
legend("Elem. C", "Elem. D");
grid;
xlim([0, Ts]);
ylabel('\sigma_x [MPa]'); xlabel('t [s]');

print("tresp_sigx.svg", "-dsvg");

```

Finally, we compute the displacements at all nodal points and export them so that the deformation can be animated in Gmsh. The output is limited to selected time steps to avoid a large amount of output data. At last, the component is saved in file `tresp.bin` which is later used for a restart.

```

# Perform backtransformation and export displacements

tb = 0 : 5 * dt : Ts;
beam = mfs_back(beam, "transresp", "disp", 1, tb);
mfs_export("tresp.dsp", "msh", beam, "transresp", "disp");

# Save component for restart

```

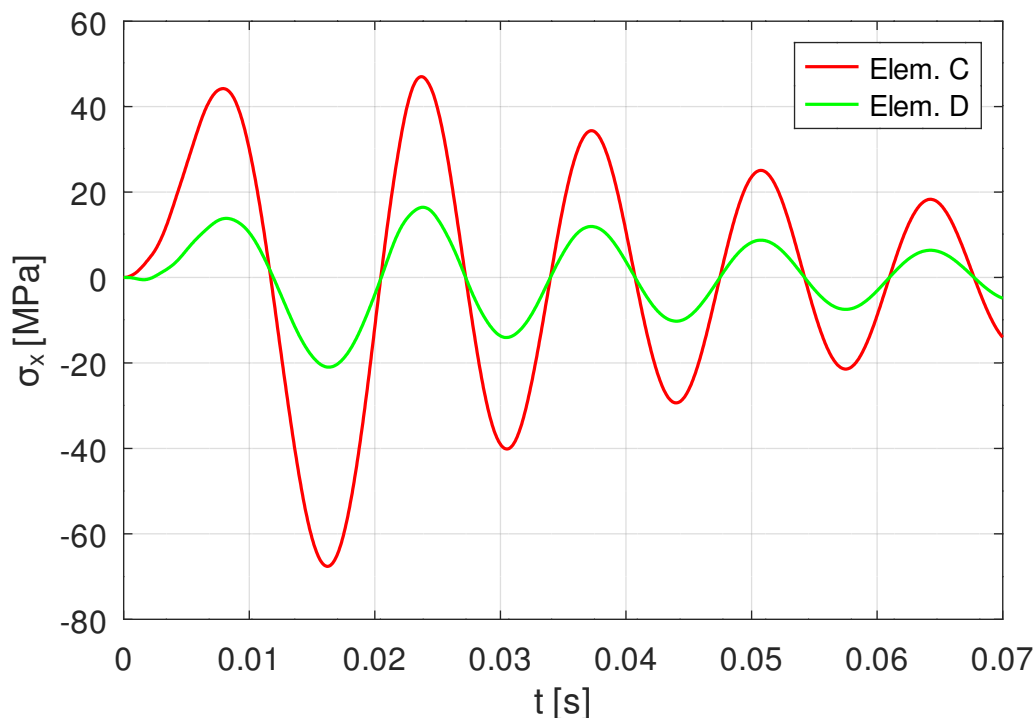


Figure 6.2-6: Box Beam, Stresses

```
save -binary tresp.bin rid eltids beam
fclose(fid);
```

In a second analysis, we perform a restart to continue the simulation. The restart starts at 0.05 s and computes the response for additional 0.1 s. The restart is chosen to overlap with the first analysis by 0.02 s so that we can check whether the results of both analyses agree within the overlapping interval.

The commands to run the restart are contained in file `restart.m`. First, the time at which to restart, the time step and the simulation time of the restart are defined. Since the response is dominated by the first vertical bending mode, we increase the time step by a factor of 10 compared to the time step used in the first analysis.

```
# Define restart parameters

t1 = 0.05;    % Time at which to restart
dt = 5e-4;    % Time step for restart
T  = 0.1;     % Simulation time

Te = T + t1;  % Total time
```

Next, we load the results of the first analysis and define the restart data. For the restart, we need to specify the number of the time step at which to restart. This time step number `stepno` is obtained by searching for `t1` in array `t` which contains the time steps of the first analysis. In addition, we assign a function `fixed` to the displacement pattern defined by load case 1. Function `fixed` defines a function  $\phi(t)$  that has a constant value of one to keep the clamped end fixed at its new position.

```
# Load results

if (! isfile("tresp.bin"))
    error("File tresp.m must be executed before\n");
endif

load tresp.bin

# Get restart data

t = mfs_getresp(beam, "transresp", "time");
stepno = lookup(t, t1);

fixed = struct("lc", 1, "func", "fixed");
```

Now we perform the transient response analysis. The keyword "**transresp**" indicates that the following argument specifies the time step of a transient response analysis which defines the initial conditions. The argument is

an array containing the result case identifier and the number of the time step. Here, the result case identifier is 1. The keyword **"rcase"** indicates that the following number defines the result case identifier of the current transient response analysis, i.e. the results of the restart are stored as result case 2.

```
# Restart transient response analysis
```

```
beam = mfs_transresp(beam, [dt, T], "transresp", [1, stepno],  
                    "load", fixed, "rcase", 2);
```

Finally, we plot the results at point *B* together with the results of the first analysis. Figure 6.2-7 shows that in the overlapping time interval, i.e. from 0.05 s to 0.07 s, the results are identical.

```
# Plot results at point B
```

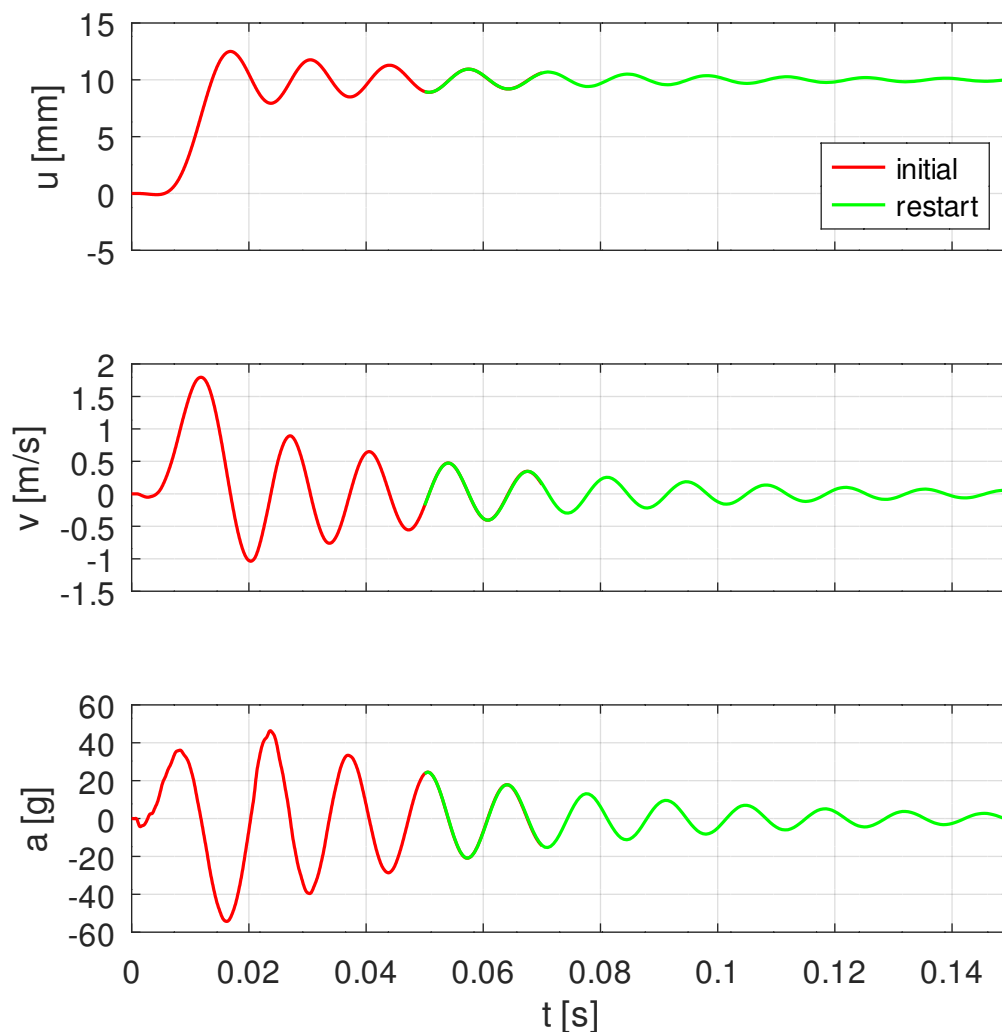


Figure 6.2-7: Box Beam, Results from Restart

```

ridB = {"Point_B", 3};

tr = t1 + mfs_getresp(beam, "transresp", "time", 2);

u = mfs_getresp(beam, "transresp", "disp", ridB, 1);
ur = mfs_getresp(beam, "transresp", "disp", ridB, 2);
v = mfs_getresp(beam, "transresp", "velo", ridB, 1) / 1000;
vr = mfs_getresp(beam, "transresp", "velo", ridB, 2) / 1000;
a = mfs_getresp(beam, "transresp", "acce", ridB, 1) / 9810;
ar = mfs_getresp(beam, "transresp", "acce", ridB, 2) / 9810;

figure(1, "position", [100, 200, 1000, 800],
      "paperposition", [0, 0, 15, 15]);
subplot(3, 1, 1)
plot(t, u, tr, ur);
legend("initial", "restart", "location", "southeast");
axis("labely");
grid; xlim([0, Te]);
ylabel('u [mm]');
subplot(3, 1, 2)
plot(t, v, tr, vr);
axis("labely");
grid; xlim([0, Te]);
ylabel('v [m/s]');
subplot(3, 1, 3)
plot(t, a, tr, ar);
grid; xlim([0, Te]);
ylabel('a [g]'); xlabel('t [s]');
print("restart.svg", "-dsvg");

```

### Transient Results from a Frequency Response Analysis

Also in this example, the transient response can be computed from the frequency response. In this example, instead of performing an inverse Fourier transformation of frequency response functions, we use the second form of function `mfs_freq2time` to compute results of class `"transresp"` from results of class `"freqresp"` and store them in the component.

The commands to run this analysis are contained in file `fresp.m`. The file begins with the same commands as in Example 6.1.

```

# Define excitation parameters

T = 0.02; % Duration of prescribed motion

# Define simulation parameters

fc = 400; % Cut-off frequency
df = 1; % Frequency resolution
dtb = 1e-4; % Time step for backtransformation
Tp = 0.07; % Time to plot

# Compute parameters for Fourier transform

```

```

dt = 1 / (2 * fc); % Time step
Ts = 1 / df;      % Duration of time series

# Get component with results

if (! isfile("modes.bin"))
    error("File tresp.m must be executed before\n");
endif

load modes.bin

```

We then perform a frequency response analysis and plot, as an example, the transfer matrix of the accelerations. This transfer matrix can be seen in Figure 6.2-8. The accelerations at the zero frequency are zero and must therefore be excluded from a semi-logarithmic plot.

```

# Perform frequency response analysis

f = 0 : df : fc; nfreq = length(f);
beam = mfs_freqresp(beam, f, "nband", 0);

# Plot transfer functions

rid = {"Point_A", 3; "Point_B", 3};

H = mfs_getresp(beam, "freqresp", "acce", rid) / 9810;

```

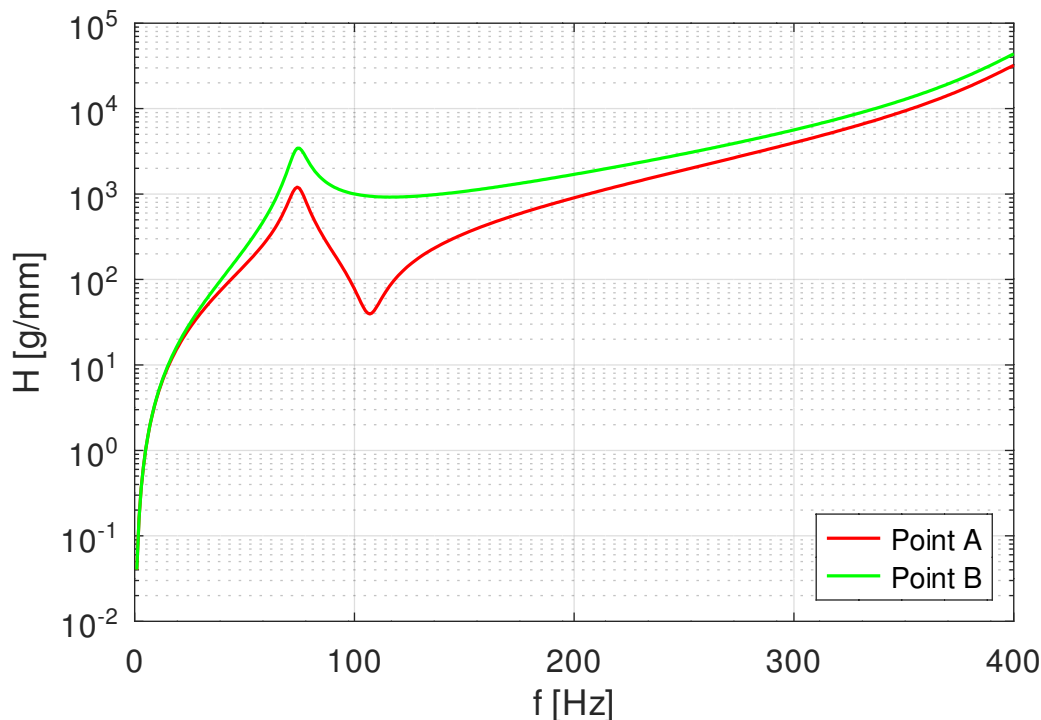


Figure 6.2-8: Box Beam, Transfer Matrix of Accelerations

```
figure(1, "position", [100, 100, 600, 400],
      "paperposition", [0, 0, 15, 10]);
semilogy(f(2 : end), abs(H(:, 2 : end)));
legend("Point A", "Point B", "location", "southeast");
grid;
xlabel('f [Hz]'); ylabel('H [g/mm]');
print("fresp_tf.svg", "-dsvg");
```

Next, we determine the Fourier transform of the prescribed displacements. First, we use function **motion** to compute the values of function  $\phi(t)$  and its first and second derivative. The analytical Fourier transform of  $\phi(t)$  contains the Dirac distribution and a term that is inversely proportional to the frequency. Therefore, its numerical computation is difficult. Instead, we compute the Fourier transform of the acceleration and divide it by  $(2\pi i f)^2$ . Of course, the zero frequency must be excluded. Setting the value at the zero frequency

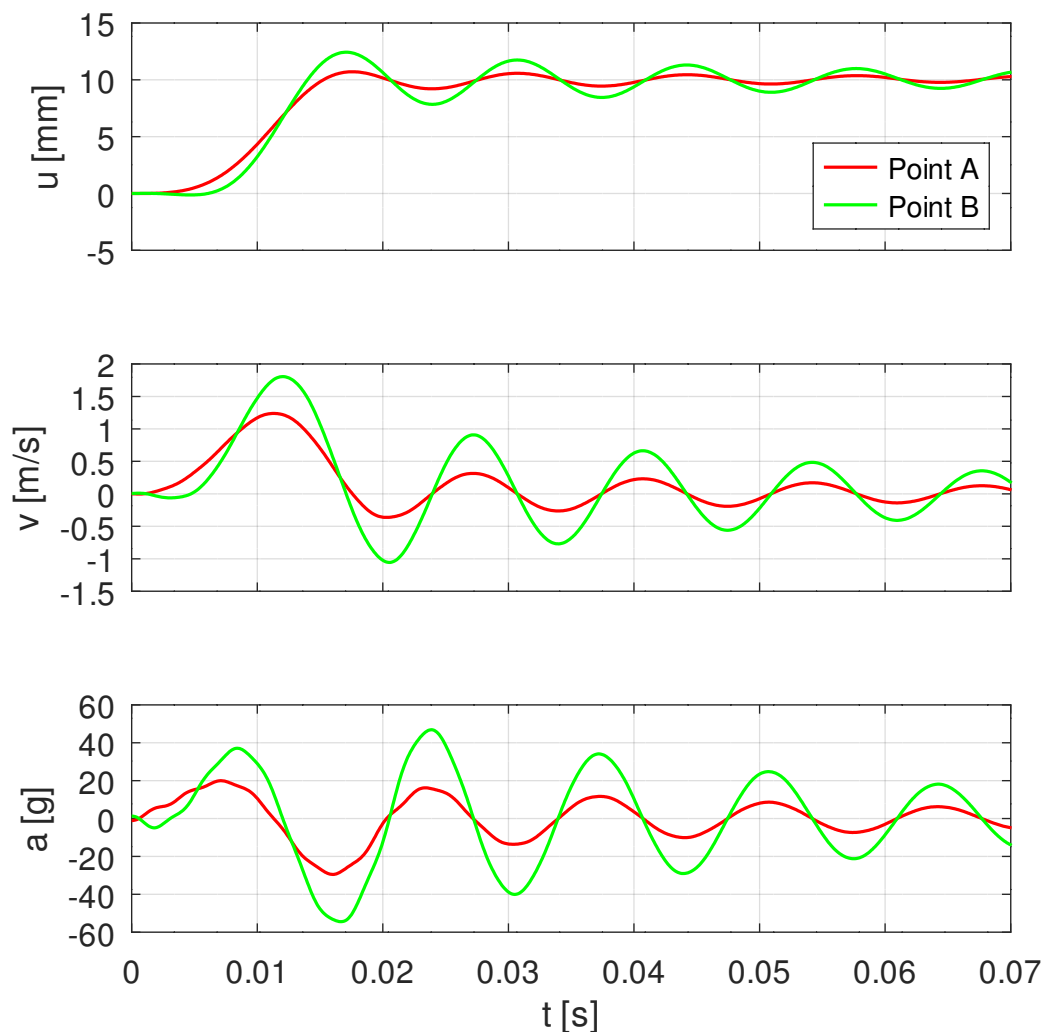


Figure 6.2-9: Box Beam, Transient Results from Frequency Response Analysis

to zero leads to a baseline drift in the displacements and velocities that must be corrected later.

```
# Get Fourier transform of prescribed displacements

t = 0 : dt : Ts;
uva = motion(t, T);
Az = fft(uva(3, :)) * dt;
wq = (2 * pi * i * f).^2;
Uz(2 : nfreq) = Az(2 : nfreq) ./ wq(2 : nfreq);

figure(2, "position", [300, 100, 750, 500]);
plot(f, abs(Uz));
grid; xlim([0, fc]);
xlabel('f [Hz]'); ylabel('|Uz| [mm/Hz]');
```

The second form of function **mfs\_freq2time** has as input arguments the component, the spectrum, an identifier for the result case of class "**trans-resp**", the time step and a flag controlling the baseline correction.

Structure **spectrum** assigns spectra to the load cases of the frequency response analysis. Field **df** defines the frequency increment. Field **lc** is an array of load case identifiers. In our case there is only load case 1. Field **spec**

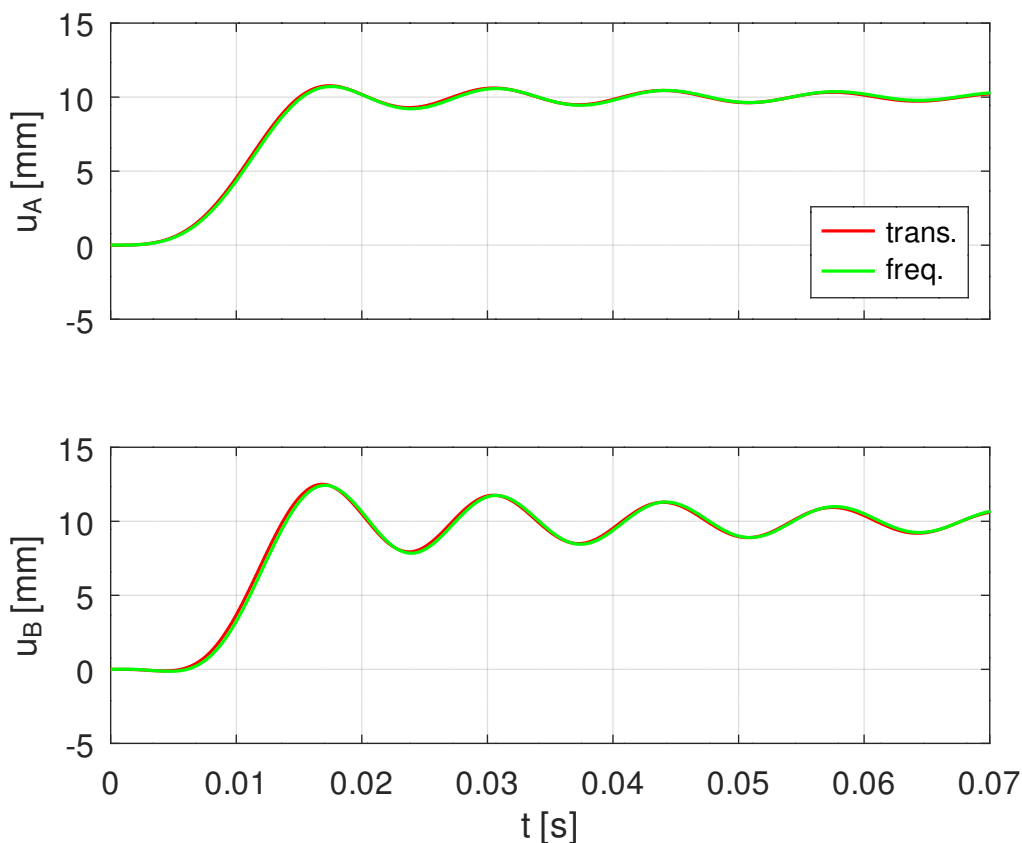


Figure 6.2-10: Box Beam, Comparison of Displacements

contains the spectra, i.e. the Fourier transforms of the loads. Each row corresponds to one load case. Here it consists of one row only.

The last argument controls the baseline correction. A value of 2 requests to add a linear function of time so that the displacements and the velocities at  $t = 0$  s are zero. The default of this flag is 0, which means that no correction is made.

```
# Perform transformation to time domain

spectrum = struct("df", df, "lc", 1, "spec", Uz);
beam = mfs_freq2time(beam, spectrum, 3, dtb, 2);
```

The transient results are now stored as result case 3 of class **"transresp"** in component **beam** and can be further processed like results of a transient response analysis. The following commands generate e.g. diagrams of the transient displacements, velocities and accelerations at points *A* and *B*. These diagrams can be seen in Figure 6.2-9.

Finally, the results are saved in file **fresp.csv** for comparison with the results from a transient response analysis.

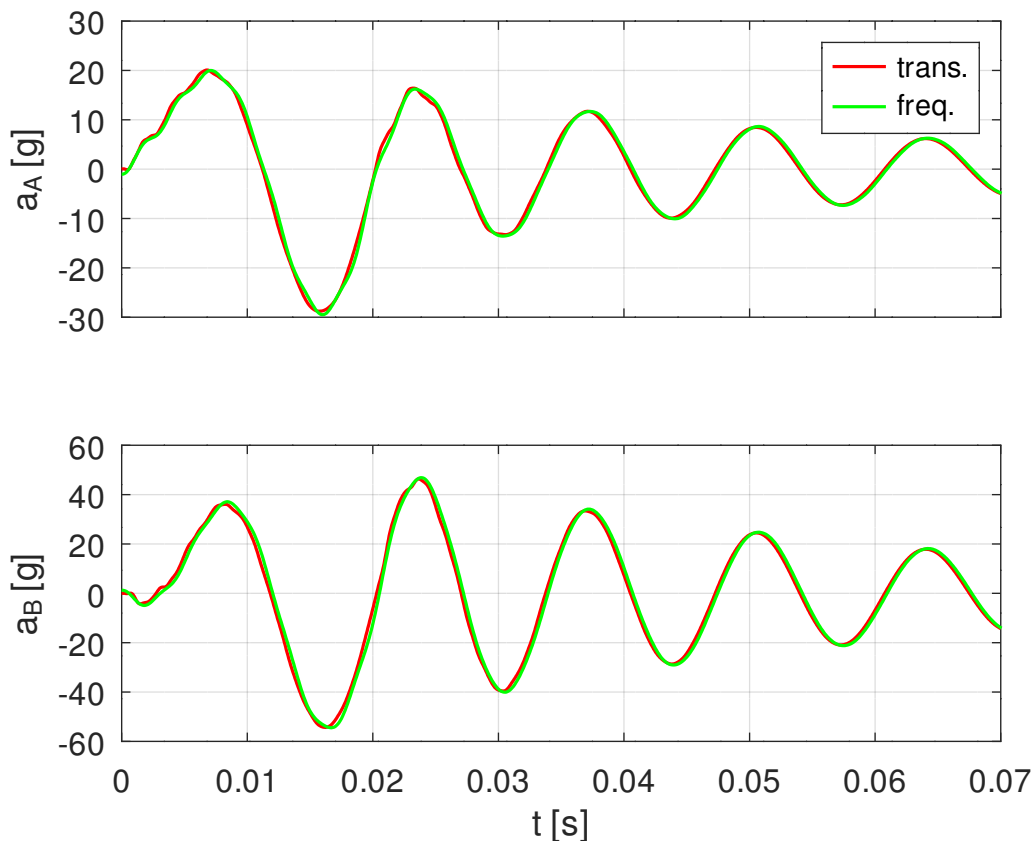


Figure 6.2-11: Box Beam, Comparison of Accelerations



```
# Plot time histories

t = mfs_getresp(beam, "transresp", "time", 3);
u = mfs_getresp(beam, "transresp", "disp", rid, 3);
v = mfs_getresp(beam, "transresp", "velo", rid, 3) / 1000;
a = mfs_getresp(beam, "transresp", "acce", rid, 3) / 9810;

figure(4, "position", [500, 200, 1000, 800],
       "paperposition", [0, 0, 15, 15]);
subplot(3, 1, 1)
plot(t, u);
legend("Point A", "Point B", "location", "southeast");
axis("labely");
grid; xlim([0, Tp]);
ylabel('u [mm]');
subplot(3, 1, 2)
plot(t, v);
axis("labely");
grid; xlim([0, Tp]);
ylabel('v [m/s]');
subplot(3, 1, 3)
plot(t, a);
grid; xlim([0, Tp]);
ylabel('a [g]'); xlabel('t [s]');
print("fresp.svg", "-dsvg");

# Save results

dlmwrite("fresp.csv", [t; u; v; a], "precision", 5);
```

Figures 6.2-10 and 6.2-11 compare the displacements and the accelerations, respectively, obtained from the transient response analysis with those from the frequency response analysis. It can be observed that the results agree very well.