Examples Manual

Mefisto 2.8

Volume 3: Aeroelasticity

Contents

1 Introduction....2
2 Steady Aeroelasticity....3
2.1 Gull Wing....3
2.2 Swept Wing....28
3 Trim Analysis....61
3.1 Glider....61
4 Flutter Analysis....104
4.1 Wing....104
5 Frequency Response Analysis....128
5.1 Glider....128

1 Introduction

The examples presented in this manual demonstrate how to perform an aeroelastic analysis using the Mefisto functions. It is assumed that you are already familiar with solid mechanics and aerodynamics and that you know how to use Gmsh for post-processing.

If you are new to Mefisto or to Gmsh, you are recommended to first study some of the examples of solid mechanics as described in Volume 1 of the Examples Manual, and of aerodynamics, described in Volume 2 of the Examples Manual.

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

`aeroelastic`	`statresp`	Steady aeroelasticity
	`trim`	Trim analysis
	`flutter`	Flutter analysis
	`freqresp`	Frequency response analysis

Some of the examples use script `iniplot.m` located in directory `exa`, so this directory must be included in the search path. If you installed Mefisto using script `install.m` nothing need be done. Otherwise, you have to add a corresponding **`addpath`** command to file `.octaverc`.

Script `iniplot.m` defines six colours (red, green, blue, magenta, brown, light blue) to be used with the plots. It also defines the default format and extension of the files created with the **`print`** command.

2 Steady Aeroelasticity

2.1 Gull Wing

Summary

Directory:	`exa/aeroelastic/statresp/gull_wing`
Objectives:	• learn how to define a simple aeroelastic model • learn how to check the quality of the splines • learn how to perform a divergence analysis • learn how to run a steady aeroelastic analysis of a clamped wing • learn how to post-process results of an aeroelastic analysis
Elements:	**b2**
Method:	Vortex-Lattice
Functions:	**mfs_line, mfs_beamsection, mfs_airfoil, mfs_new, mfs_export, mfs_stiff, mfs_mass, mfs_massproperties, mfs_splines, mfs_freevib, mfs_statresp, mfs_diverg, mfs_results, mfs_print, mfs_getresp, mfs_xydata, mfs_merge, mfs_transfer**

Problem Description

Perform an aeroelastic analysis of the gull wing shown in Figures 2.1-1 and 2.1-2. Compute the divergence speed and compute the aerodynamic pressure and the displacements for the following two configurations:

1. Angle of attack: 2°; aileron angle: 0°
2. Angle of attack: 2°; aileron angle: 2°

The flight velocity is $144\ \mathrm{km/h}$ and the mass density of the air is $1.21\ \mathrm{kg/m^3}$.

Compare the results of the flexible wing with those of the rigid wing.

The solid structure is represented by a beam model that is shown in Figure 2.1-1. The local z_E-axis of all beams is perpendicular to the wing surface and points upwards.

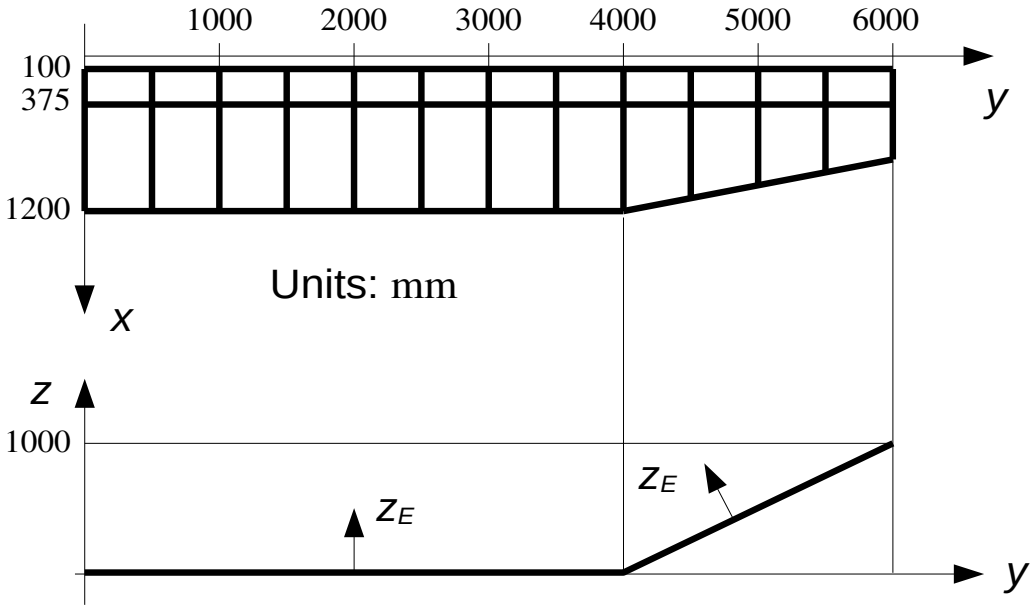


Figure 2.1-1: Gull Wing, Solid Structure

The beams have the following cross sections:

- The front spar ($x = 100$ mm) and the rear spar ($x = 1200$ mm) have `I`-sections with a height of 100 mm, a width of 30 mm and a wall thickness of 2 mm.
- The cross section of the main spar ($x = 375$ mm) is a thin-walled rectangular box with a height of 180 mm, a width of 150 mm and a wall thickness of 2 mm.
- The ribs have an `I`-section with a height of 120 mm, a width of 40 mm and a wall thickness of 1 mm.

The material is aluminium with a Young's modulus of 70000 MPa, a Poisson's ratio of 0.34 and a mass density of 2700 kg/m^3.

The wing is clamped at the wing root. At the front and rear spars, only the translational degrees of freedom are constrained. At the main spar, also the rotational degrees of freedom are constrained.

Figure 2.1-2 shows the geometry of the aerodynamic model. The wing has a NACA 43012 airfoil. The flap chord ratio of the aileron is 20 %.

Model Definition

In an aeroelastic analysis, three different models need be defined, namely a solid model, an aerodynamic model and an aeroelastic model.

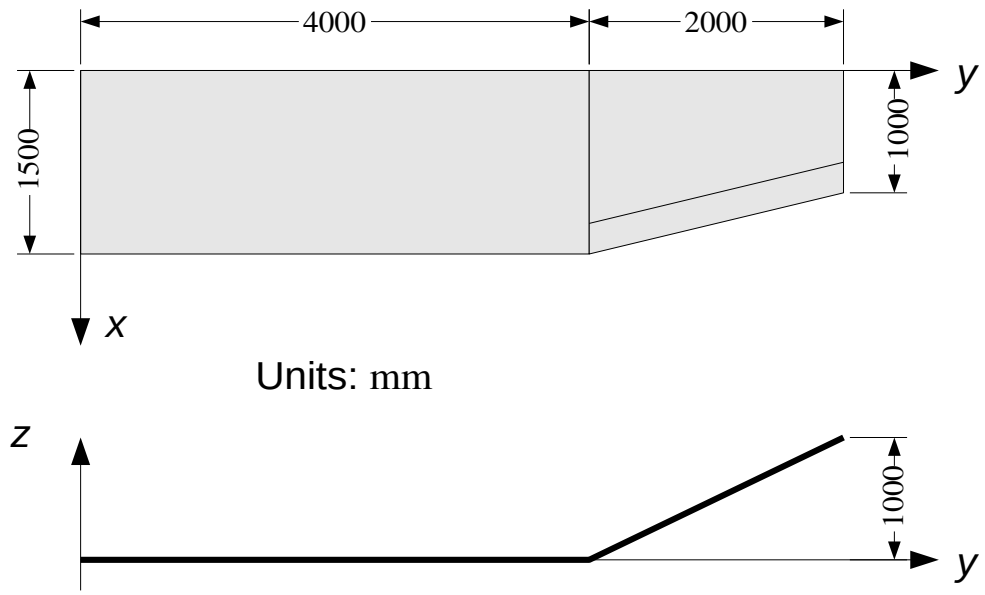


Figure 2.1-2: Gull Wing, Aerodynamic Model

The **solid model** is defined in file `solid.m`:

```
# Example: Gull wing
#
# Solid model:
#  - Definition of the solid model
#  - Computation of first five normal modes to check the model
#  - Storage of solid model including node sets in binary file
#    solid.bin
#
# ------------------------------------------------------------

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

# Data (N, mm, s)
# ---------------

# Geometry

  x1  =    100;  % x-position of front spar
  x2  =    375;  % x-position of main spar
  x3  =   1200;  % x-position of rear spar at wing root
  x4  =    800;  % x-position of rear spar at wing tip

  ya  =   4000;  % y-position of kink
  yt  =   6000;  % y-position of wing tip
  zt  =   1000;  % z-position of wing tip

# Material

  E   =  70000;  % Young's modulus
```

```
  ny  =   0.34;  % Poisson's ratio
  rho = 2.7E-9;  % Mass density

# Cross sections

  frontspar = struct("type", "I",                % Front Spar
                     "b",     30, "h", 100,
                     "t",      2, "s",   2);

  mainspar = struct("type", "box",               % Main Spar
                    "styp", "thin", "b", 150,
                    "h",       180, "t",   2);

  rearspar = struct("type", "I",                 % Rear Spar
                    "b",     30, "h", 100,
                    "t",      2, "s",   2);

  ribs     = struct("type", "I",                 % Ribs
                    "b",     40, "h", 120,
                    "t",      1, "s",   1);

# Model Definition
# ----------------

# Model type

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

# Material

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

# Front spar

  geom   = mfs_beamsection(frontspar.type, frontspar.b,
                           frontspar.h, frontspar.t,
                           frontspar.s);
  geom.v = [0, 0, 1];

  nodes(1) = struct("id",  101, "coor", [ 100,    0,    0]);
  nodes(2) = struct("id",  901, "coor", [ 100, 4000,    0]);
  nodes(3) = struct("id", 1301, "coor", [ 100, 6000, 1000]);

  idnew = 201 : 100 : 801;
  idelt = 1 : 8;

  [nodes, elem1] = mfs_line(nodes, 101, 901, idnew, idelt,
                            "b2", geom, mat);

  idnew = 1001 : 100 : 1201;
  idelt = 9 : 12;

  [nodes,elem2] = mfs_line(nodes, 901, 1301, idnew, idelt,
                           "b2", geom, mat);
```

```
# Main spar

  geom   = mfs_beamsection(mainspar.type, mainspar.styp,
                           mainspar.b, mainspar.h,
                           mainspar.t);
  geom.v = [0, 0, 1];

  n = length(nodes);

  nodes(++n) = struct("id",  111, "coor", [ 375,    0,    0]);
  nodes(++n) = struct("id",  911, "coor", [ 375, 4000,    0]);
  nodes(++n) = struct("id", 1311, "coor", [ 375, 6000, 1000]);

  idnew = 211 : 100 : 811;
  idelt = 13 : 20;

  [nodes, elem3] = mfs_line(nodes, 111, 911, idnew, idelt,
                            "b2", geom, mat);

  idnew = 1011 : 100 : 1211;
  idelt = 21 : 24;

  [nodes,elem4] = mfs_line(nodes, 911, 1311, idnew, idelt,
                           "b2", geom, mat);

# Rear spar

  geom   = mfs_beamsection(rearspar.type, rearspar.b,
                           rearspar.h, rearspar.t,
                           rearspar.s);
  geom.v = [0, 0, 1];

  n = length(nodes);

  nodes(++n) = struct("id",  141, "coor", [ 1200,    0,    0]);
  nodes(++n) = struct("id",  941, "coor", [ 1200, 4000,    0]);
  nodes(++n) = struct("id", 1341, "coor", [  800, 6000, 1000]);

  idnew = 241 : 100 : 841;
  idelt = 25 : 32;

  [nodes, elem5] = mfs_line(nodes, 141, 941, idnew, idelt,
                            "b2", geom, mat);

  idnew = 1041 : 100 : 1241;
  idelt = 33 : 36;

  [nodes,elem6] = mfs_line(nodes, 941, 1341, idnew, idelt,
                           "b2", geom, mat);


  elemh = [elem1, elem2, elem3, elem4, elem5, elem6];

# Ribs in front of main spar
```

```
  geomi   = mfs_beamsection(ribs.type, ribs.b, ribs.h,
                            ribs.t, ribs.s);
  geomi.v = [0, 0, 1];

  geoma   = mfs_beamsection(ribs.type, ribs.b, ribs.h,
                            ribs.t, ribs.s);
  geoma.v = [0, -1 , 2];

  ide = length(elemh) + 1;
  id1 = 101; id2 = 111;

  for n = 1 : 9
      elemr(n) = struct("id",   ide++, "nodes", [id1, id2],
                        "type",  "b2", "geom",  geomi,
                        "mat",    mat);
      id1 += 100; id2 += 100;
  endfor

  for n = 10 : 13
      elemr(n) = struct("id",   ide++, "nodes", [id1, id2],
                        "type",  "b2", "geom",  geoma,
                        "mat",    mat);
      id1 += 100; id2 += 100;
  endfor

# Ribs behind main spar

  id1 = 111; id2 = 141; idnew = [121, 131];

  for n = 1 : 9
      idelt = ide : ide + 2;
      [nodes, elemx] = mfs_line(nodes, id1, id2, idnew, idelt,
                                "b2", geomi, mat);
      id1 += 100; id2 += 100; idnew += 100;
      ide += 3;
      elemr = [elemr, elemx];
  endfor

  for n = 1 : 4
      idelt = ide : ide + 2;
      [nodes, elemx] = mfs_line(nodes, id1, id2, idnew, idelt,
                                "b2", geoma, mat);
      id1 += 100; id2 += 100; idnew += 100;
      ide += 3;
      elemr = [elemr, elemx];
  endfor

  solid.nodes     = nodes;
  solid.elements  = [elemh, elemr];

# Node sets (needed to define the splines)

  nset.inner_wing = [ 101 : 100 : 901, 111 : 100 : 911, ...
                      121 : 100 : 911, 131 : 100 : 931, ...
                      141 : 100 : 941 ];
```

```
  nset.outer_wing = [ 901 : 100 : 1301, 911 : 100 : 1311, ...
                      921 : 100 : 1321, 931 : 100 : 1331, ...
                      941 : 100 : 1341 ];

  solid.nset = nset;

# Constraints

  prescribed = struct("id",   {  101,   111,   141},
                      "dofs", {1 : 3, 1 : 6, 1 : 3});
  solid.constraints.prescribed = prescribed;
```

The resulting mesh with its node numbers can be seen in Figure 2.1-3.

The **aerodynamic model** consists of three lifting surfaces, one for the inner wing, one for the outer wing and one for the aileron (cf. Figure 2.1-4). It is defined in file `rigid.m`:

```
# Example: Gull wing
#
# Aerodynamic analysis of rigid wing
#  - Definition of the aerodynamic model
#  - Analysis of two configurations
#  - Storage of aerodynamic model including pressure in binary
```

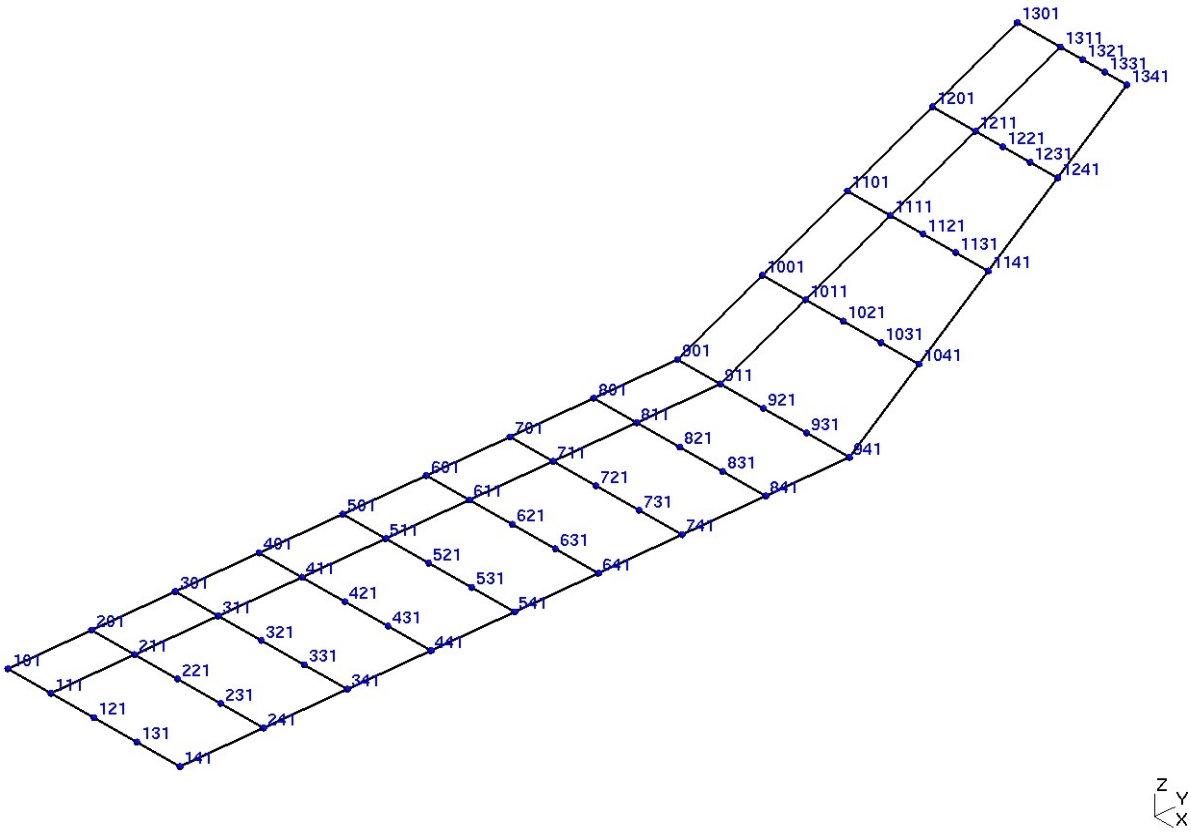


Figure 2.1-3: Gull Wing, Mesh of Solid Structure

```
#    file aero.bin
#  - Storage of xy-plot data in binary file rigid.plt
#
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

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

# Data (N, mm, s)
# --------------

  ya = 4000;  % y-position of aileron in mm
  yt = 6000;  % y-position of wing tip in mm
  zt = 1000;  % z-position of wing tip in mm

  cr = 1500;  % Chord length at wing root
  ct = 1000;  % Chord length at wing tip
  fr  =  0.2;  % Flap chord ratio of aileron

  nxi =  50;  % No. of panels in x-direction, inner wing
  nxo =  40;  % No. of panels in x-direction, outer wing
  nxa =  10;  % No. of panels in x-direction, aileron

  nyi =  25;  % No. of panels in y-direction, inner wing
  nyo =  30;  % No. of panels in y-direction, outer wing

  alpha =        2;   % Angle of attack in degrees
  eta   =   {0, 2};   % Aileron angles in degrees
  v     =     40E3;   % Flight veloctiy in mm/s
  rho   = 1.21E-12;   % Mass density of air in t/mm^3

  frc   = 1 - fr;
```

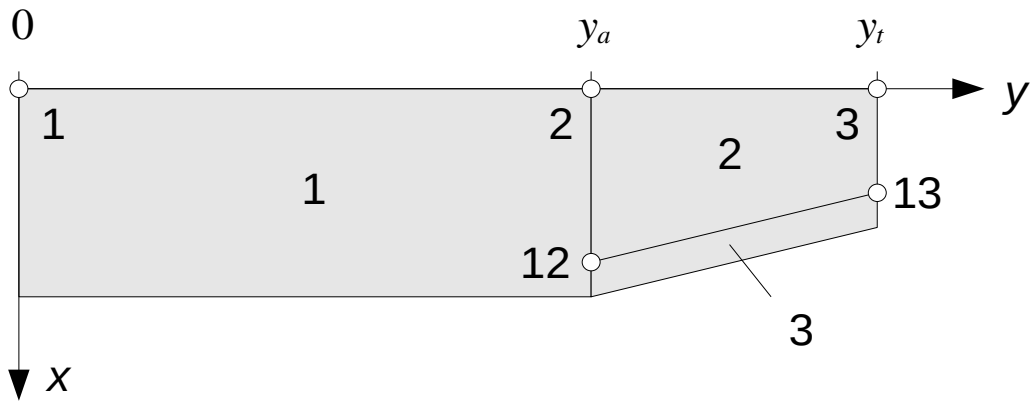


Figure 2.1-4: Gull Wing, Lifting Surfaces

```
# Model definition
# ----------------

# Model type, subtype and symmetry

  model = struct("type", "aero", "subtype", "vlm", "symy", 0);

# Airfoil data

  ppi = mfs_airfoil("NACA", 5, 30);
  ppo = mfs_airfoil("NACA", 5, 30, 0, frc);
  ppa = mfs_airfoil("NACA", 5, 30, frc, 1);

# Points on leading edge of wing

  points(1 : 3) = struct("id", {1, 2, 3},
                         "coor", {[0, 0, 0], [0, ya, 0], ...
                                  [0, yt, zt]});

# Points on leading edge of aileron

  points(4 : 5) = struct("id", {12, 13},
                         "coor", {[frc * cr, ya,  0], ...
                                  [frc * ct, yt, zt]});

# Lifting surfaces of inner wing, outer wing, aileron

  lsf = struct("id",     {1, 2, 3},
               "points", {[1, 2], [2, 3], [12, 13]},
               "chord",  {cr, frc * [cr, ct], fr * [cr, ct]},
               "camber", {ppi, ppo, ppa},
               "nx",     {nxi, nxo, nxa},
               "ny",     {nyi, nyo, nyo},
               "typex",  "linear",
               "typey",  {"linear", "cos>", "cos>"});

# Aileron is control surface

  controls = struct("name", "aileron", "ls", 3);

# Configurations

  qdyn = 0.5 * rho * v^2;

  for l = 1 : 2
      cfgnam{l} = sprintf("Conf. %d: alpha = %5.2f, eta = %5.2f",
                          l, alpha, eta{l});
  endfor

  config = struct("name", cfgnam, "qdyn", qdyn,
                  "alpha", alpha, "aileron", eta);

# Add definitions to model

  model.points   = points;
```

```
model.ls       = ls;
model.controls = controls;
model.config   = config;
```

Figure 2.1-5 shows the resulting discretization of the lifting surfaces.

The **aeroelastic model** combines the solid model and the aerodynamic model. We have to specify the solid component and the aerodynamic component and define the splines. This is done in file `splines.m`:

```
# Example: Gull wing
#
# Splines
#  - Definition of aeroelastic model
#  - Transfer of aerodynamic loads to solid model
#  - Computation of displacements
#  - Storage of aeroelastic component in binary file
#    aeroelastic.bin
#
# Files needed: solid.bin, aero.bin
#
# ------------------------------------------------------------

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

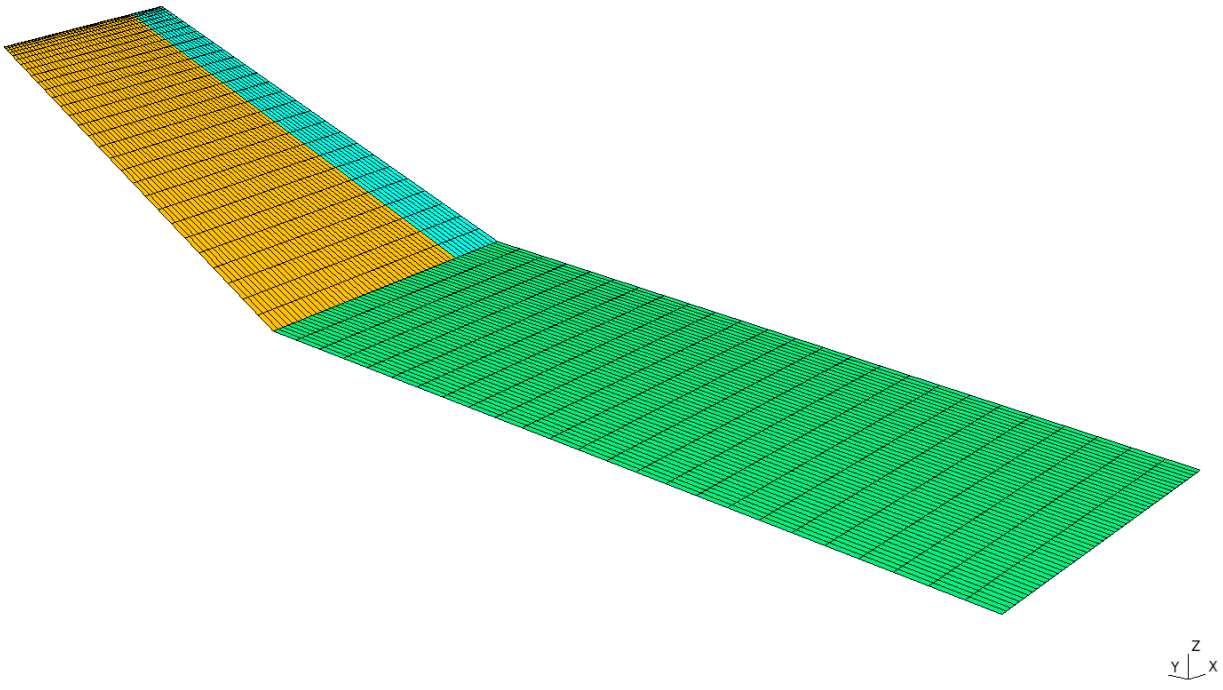


Figure 2.1-5: Gull Wing, Aerodynamic Mesh

```
# Data
# ----

  nsi = 8;     % Number of spline breaks of outer wing
  nso = 4;     % Number of spline breaks of inner wing
```

The number of spline breaks controls the quality of the splines that connect the solid and the aerodynamic component. The splines use a least squares fitting to approximate the solid displacements normal to the lifting surfaces. The larger the number of spline breaks, the better the solid displacements are approximated. However, if the number of spline breaks is too large, the problem becomes ill-conditioned.

The inner wing has nine rows of structural nodes and the outer wing has five rows. Thus, the number of spline breaks of the inner wing should not be larger than nine and that of the outer wing should not be larger than five. We use eight spline breaks for the inner wing and four for the outer wing.

The model type of an aeroelastic component is **`"aeroelastic"`**. There is no need to define a subtype because the subtypes are inherited from the solid and the aerodynamic component. The subtype of the solid component must be **`"3d"`**. The solid and the aerodynamic components have been saved to files `solid.bin` and `aero.bin` in previous runs. They are loaded and assigned to fields **`solid`** and **`aero`** of structure **`model`**.

```
# Aeroelastic model
# -----------------

  load solid.bin
  load aero.bin

  model = struct("type", "aeroelastic",
                 "solid", wings, "aero", winga);
```

Next, we define the splines. The only spline type currently available is the torsion-bending spline. It approximates the deformation of the lifting surface by a combination of bending and torsion.

One spline is needed for each lifting surface. Lifting surface 1, representing the inner wing, is connected to the nodal points of the inner wing. Lifting surfaces 2 and 3, representing the outer wing with the aileron, are connected to the nodal points of the outer wing.

```
# Splines

  model.splines = struct("id",     {1, 2, 3},
                         "type",   "tb",
                         "lsid",   {1, 2, 3},
                         "nodes", {"inner_wing", ...
```

```
                                    "outer_wing", ...
                                    "outer_wing"},
                          "data",  {struct("nbreaks", nsi), ...
                                    struct("nbreaks", nso), ...
                                    struct("nbreaks", nso)});
```

Analysis

The complete analysis is subdivided into five steps (cf. Figure 2.1-6):

1. File `solid.m` defines the solid model and computes the first five normal modes to check the solid model.
2. File `rigid.m` defines the aerodynamic model including the configurations and computes the pressure on the rigid wing.
3. File `splines.m` defines the aeroelastic model. To check the splines, the aerodynamic loads computed in step 2 are transferred to the solid model defined in step 1, and the displacements are computed.
4. File `diverg.m` performs the divergence analysis.
5. File `flexible.m` computes the aerodynamic pressure on the flexible wing and the displacements of the wing.

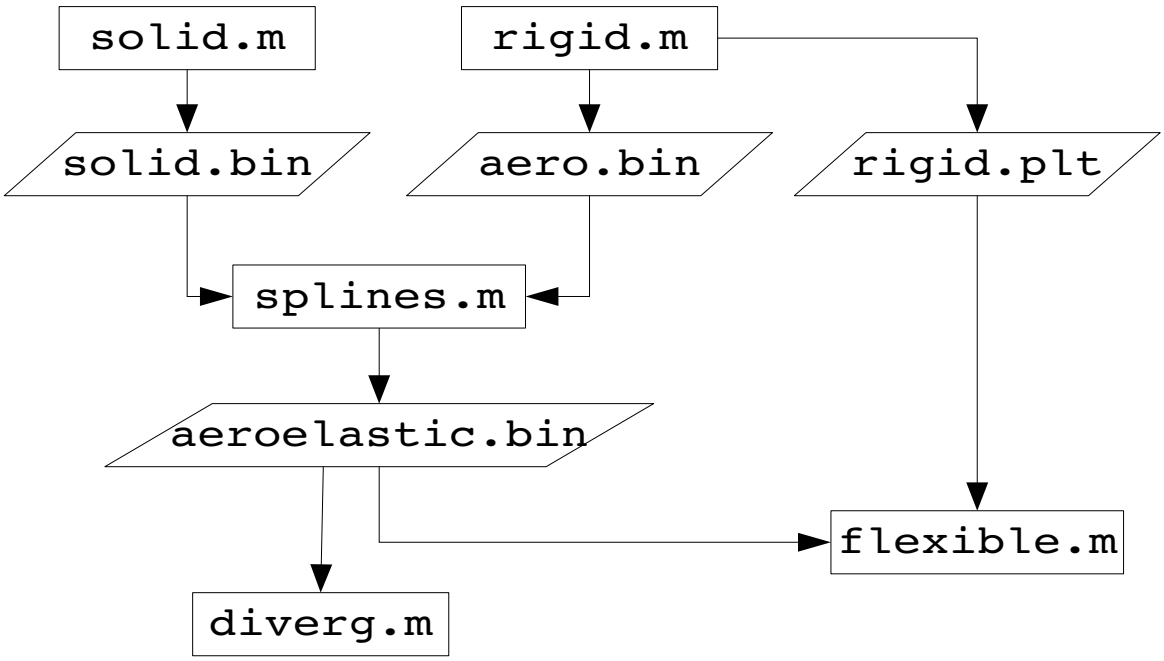


Figure 2.1-6: Gull Wing, Computational Steps and Data Flow

Step 1: Solid Model and Normal Modes

As described above GNU Octave script `solid.m` defines the solid model. The script continues with a normal modes analysis:

```
# Analysis
# --------

# Create and export component

  wings = mfs_new(fid, solid);
  mfs_export("solid.msh", "msh", wings, "mesh", "mesh", "axes");

# Compute stiffness and mass matrices

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

# Save the solid component

  save solid.bin wings

# Compute and export the normal modes

  wings = mfs_freevib(wings, 5);
  mfs_print(fid, wings, "modes", "freq");
  mfs_export("modes.dsp", "msh", wings, "modes", "disp");

  fclose(fid);
```

Following the creation of the component and the computation of the matrices the component is saved to file `solid.bin`.

The output file contains the following information:

```
Mefisto 2.7: Building new component from input "solid"

  Model Type = solid, Model Subtype = 3d

  Number of nodes      =    65,  Number of elements =    88
  Number of element types =  1
  Number of global     degrees of freedom =   390
  Number of local      degrees of freedom =   378
  Number of prescribed degrees of freedom =    12
  Number of dependent  degrees of freedom =     0


Mass properties of component "wings"

  Coordinates of reference point:      0.0000,     0.0000,      0.0000

  Rigid body mass matrix:

     3.9872e-02 -4.6374e-20 -1.3781e-20 -4.5716e-17  6.9525e+00 -1.2082e+02
     3.7329e-20  3.9872e-02  6.4080e-19 -6.9525e+00 -1.7724e-16  1.9225e+01
     1.8827e-20  5.9280e-19  3.9872e-02  1.2082e+02 -1.9225e+01  5.2001e-17
     7.5471e-17 -6.9525e+00  1.2082e+02  4.9551e+05 -5.6595e+04 -3.0150e+03
     6.9525e+00 -1.6932e-16 -1.9225e+01 -5.6595e+04  1.8475e+04 -3.7405e+04
    -1.2082e+02  1.9225e+01  2.6481e-16 -3.0150e+03 -3.7405e+04  5.0416e+05
```

```
  Mass =  3.9872e-02

  Inertia tensor with respect to reference point:

     4.9551e+05 -5.6595e+04 -3.0150e+03
    -5.6595e+04  1.8475e+04 -3.7405e+04
    -3.0150e+03 -3.7405e+04  5.0416e+05

  Coordinates of center of mass:   482.1608,  3030.1327,   174.3704

  Inertia tensor with respect to center of mass:

     1.2820e+05  1.6580e+03  3.3726e+02
     1.6580e+03  7.9932e+03 -1.6338e+04
     3.3726e+02 -1.6338e+04  1.2879e+05

--------------------------------------------------------------------------------

Component "wings"

Natural frequencies:

  Mode   Circ. Frequency    Frequency
  ------------------------------------
     1        26.56003       4.22716 Hz
     2        37.73625       6.00591 Hz
     3       110.43352      17.57604 Hz
     4       152.32502      24.24328 Hz
     5       159.60354      25.40169 Hz
```

Step 2: Aerodynamic Analysis of Rigid Wing

GNU Octave script `rigid.m` defines the aerodynamic model and performs an aerodynamic analysis of the rigid wing. The model definition part of the script is described above. The analysis begins with creating and exporting the component. Next, the aerodynamic analysis is performed. The component together with the mass density of the air is saved to file `aero.bin`, and the results are exported to Gmsh:

```
# Analysis
# --------

# Create and export component

  winga = mfs_new(fid, model);
  mfs_export("aero.msh", "msh", winga, "mesh");

# Aerodynamic analysis

  winga = mfs_statresp(winga);
  winga = mfs_results(winga, "statresp", "panel");

  save aero.bin winga rho

  mfs_export("rigid.prs", "msh", winga, "statresp", "pressure");
```

Then, the pressure in some wing sections is retrieved for plotting. The indices of the panel columns defining the wing sections are stored in array **`ycols`**.

```
# Pressure in selected wing sections

  ycols = [1, floor(0.8 * nyi), 3, floor(0.8 * nyo)];

  [xi(:, 1), pri(:, :, 1), y(1)] = ...
     mfs_xydata(winga, "statresp", "pressure", 1, ycols(1));
  [xi(:, 2), pri(:, :, 2), y(2)] = ...
     mfs_xydata(winga, "statresp", "pressure", 1, ycols(2));

  [xo(:, 1), pro(:, :, 1), y(3)] = ...
     mfs_xydata(winga, "statresp", "pressure", [2, 3], ycols(3));
  [xo(:, 2), pro(:, :, 2), y(4)] = ...
     mfs_xydata(winga, "statresp", "pressure",
                [2, 3], ycols(4));
```

Before plotting, pressure units are converted to kPa and length units to m. This results in a better format of the plot labels.

```
  pri = 1000 * pri;   % Pressure in kPa
  pro = 1000 * pro;

  xi  = xi / 1000;    % Lengths in m
  xo  = xo / 1000;
  y   =  y / 1000;
```

Two subplots are created, corresponding to the two configurations. Each of the two plots contains four curves corresponding to the four wing sections. The legends are built from the *y*-coordinates of the wing sections and the plot titles from the aileron deflection angle.

```
  for k = 1 : length(y)
      leg{k} = sprintf("y = %4.2f m", y(k));
  endfor
  for k = 1 : length(eta)
      header{k} = sprintf("\\eta = %2.0f\\deg", eta{k});
  endfor

  crm = cr / 1000;

  figure(1, "position", [100, 500, 750, 500],
            "paperposition", [0, 0, 14, 10]);
  subplot(1, 2, 1)   % Configuration 1
     plot(xi(:, 1), squeeze(pri(:, 1, 1)),
          xi(:, 2), squeeze(pri(:, 1, 2)),
          xo(:, 1), squeeze(pro(:, 1, 1)),
          xo(:, 2), squeeze(pro(:, 1, 2)));
     legend(leg, "location", "northeast");
     title(header{1});
     xlim([0, crm]); ylim([0, 0.9]);
     grid;
     xlabel("x [m]");ylabel("p [kPa]");
  subplot(1, 2, 2)   % Configuration 2
     plot(xi(:, 1), squeeze(pri(:, 2, 1)),
```

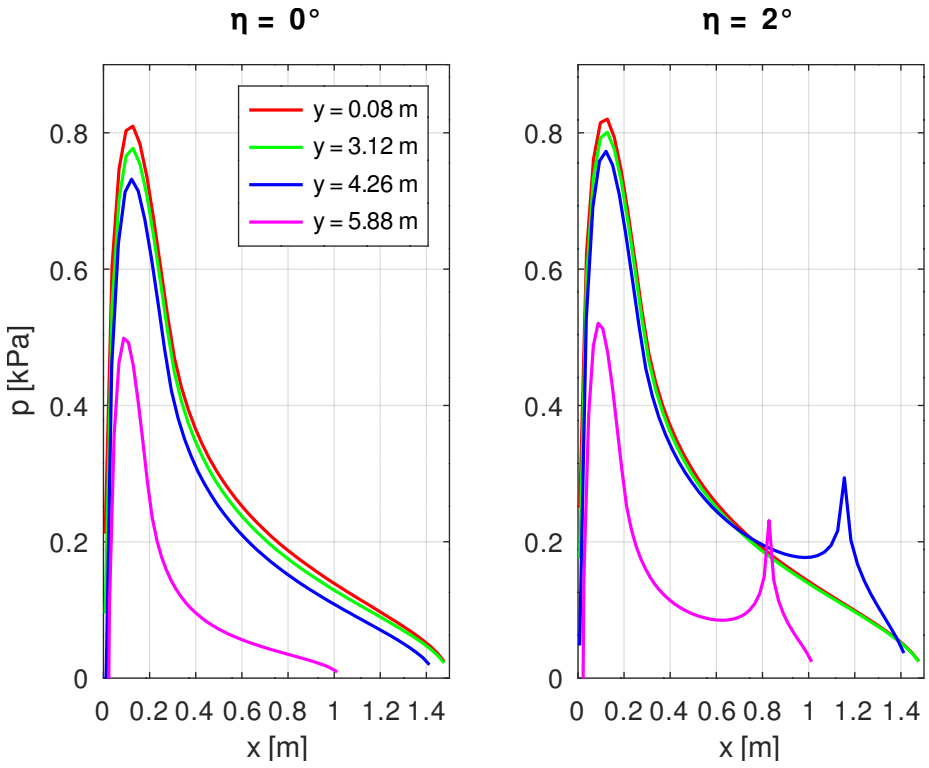


Figure 2.1-7: Gull Wing, Pressure on Rigid Wing

```
          xi(:, 2), squeeze(pri(:, 2, 2)),
          xo(:, 1), squeeze(pro(:, 2, 1)),
          xo(:, 2), squeeze(pro(:, 2, 2)));
    title(header{2});
    xlim([0, crm]); ylim([0, 0.9]);
    grid;
    xlabel("x [m]");
print(["rigid", EXT], FORMAT);

save rigid.plt ycols crm eta pri pro

fclose(fid);
```

The resulting diagrams can be seen in Figure 2.1-7.

Finally, the plot data are saved to file `rigid.plt`. The data are needed in Step 5 to compare the pressure from the flexible analysis with that of the rigid analysis.

Step 3: Splines

As described above, the aeroelastic model with the splines is defined in GNU Octave script `splines.m`. The script continues with the creation of the aeroelastic component and the computation of the splines. The aeroelastic component together with the mass density of the air is saved to file `aeroelastic.bin`.

```
# Analysis
# --------

# Create component and compute splines

  wing = mfs_new(fid, model);
  wing = mfs_splines(wing);

  save -binary aeroelastic.bin wing rho
```

Next, function **mfs_transfer** is used to transfer the aerodynamic loads from the aerodynamic component to the solid component. The first input argument of this function is the structure with the aeroelastic component. This component contains the splines but not the aerodynamic results. The second input argument is the structure with the aerodynamic component containing the aerodynamic results. The output argument is a structure with the solid component containing the stiffness matrix, the mass matrix and the loads computed from the aerodynamic pressure.

```
# Transfer aerodynamic loads from rigid aerodynamic analysis
# to the solid component and retrieve load resultants

  wings = mfs_transfer(wing, winga, "statresp", "loads");
  [FA, MA] = mfs_getresp(winga, "statresp", "aeload");
  [FS, MS] = mfs_getresp(wings, "load", "resultant");

  nc = columns(FA);

  fprintf(fid, "Load resultants of aerodynamic component: \n\n");
  for n = 1 : nc
      fprintf(fid, "  Configuration %2d:\n", n)
      fprintf(fid, "     F = [%10.3e, %10.3e, %10.3e] kN\n",
              FA(:, n) / 1000);
      fprintf(fid, "     M = [%10.3e, %10.3e, %10.3e] kNm\n",
              MA(:, n) * 1e-6);
  endfor

  fprintf(fid, "\nLoad resultants of solid component: \n\n");
  for n = 1 : nc
      fprintf(fid, "  Configuration %2d:\n", n)
      fprintf(fid, "     F = [%10.3e, %10.3e, %10.3e] kN\n",
              FS(:, n) / 1000);
      fprintf(fid, "     M = [%10.3e, %10.3e, %10.3e] kNm\n",
              MS(:, n) * 1e-6);
  endfor
```

Function **mfs_getresp** returns the resultants of the aerodynamic pressure on the aerodynamic component and of the aerodynamic loads on the solid component. The resultants are written to the output file:

```
Mefisto 2.7: Building new component from input "model"
```

```
  Model Type = aeroelastic

  Number of splines      =   3

Load resultants of aerodynamic component:

  Configuration  1:
     F = [ 0.000e+00, -2.596e-01,  2.136e+00] kN
     M = [ 5.788e+00, -9.063e-01, -9.884e-02] kNm
  Configuration  2:
     F = [ 0.000e+00, -3.310e-01,  2.343e+00] kN
     M = [ 6.690e+00, -1.039e+00, -1.493e-01] kNm

Load resultants of solid component:

  Configuration  1:
     F = [ 0.000e+00, -2.596e-01,  2.136e+00] kN
     M = [ 5.788e+00, -9.063e-01, -9.884e-02] kNm
  Configuration  2:
     F = [ 0.000e+00, -3.310e-01,  2.343e+00] kN
     M = [ 6.690e+00, -1.039e+00, -1.493e-01] kNm
```

The load resultants of the aerodynamic and the solid component are exactly identical. A difference between the load resultants would indicate severe errors in the definition of the splines.

Next, the structural displacements due to the aerodynamic loads are computed and transferred to the aerodynamic component. Now, the transfer is from the solid component to the aerodynamic component. Hence, the second input argument to function **`mfs_transfer`** is the structure with the solid component containing the displacements, and the output argument is a structure with the aerodynamic component. Subsequently, both sets of displacements are exported for post-processing with Gmsh:

```
# Compute displacements and transfer them to the aerodynamic
# component

  wings = mfs_statresp(wings);
  winga = mfs_transfer(wing, wings, "statresp", "disp");

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

  mfs_export("solid.dsp", "msh", wings, "statresp", "disp");
  mfs_export("aero.dsp", "msh", winga, "statresp", "disp");
```

Finally, both mesh files are combined into one single mesh file, and also the displacement results are combined. This allows to visualize the combined mesh and the displacements in Gmsh. Please note that the order of the result files must match the order of the corresponding mesh files, i.e. if the solid mesh is the first mesh also the solid displacements must also come first.

```
# Combine the models and the displacements

  mfs_merge("solid.msh", "aero.msh", "wing.msh", "msh");
  mfs_merge("solid.dsp", "aero.dsp", "wing.dsp", "msh");
```

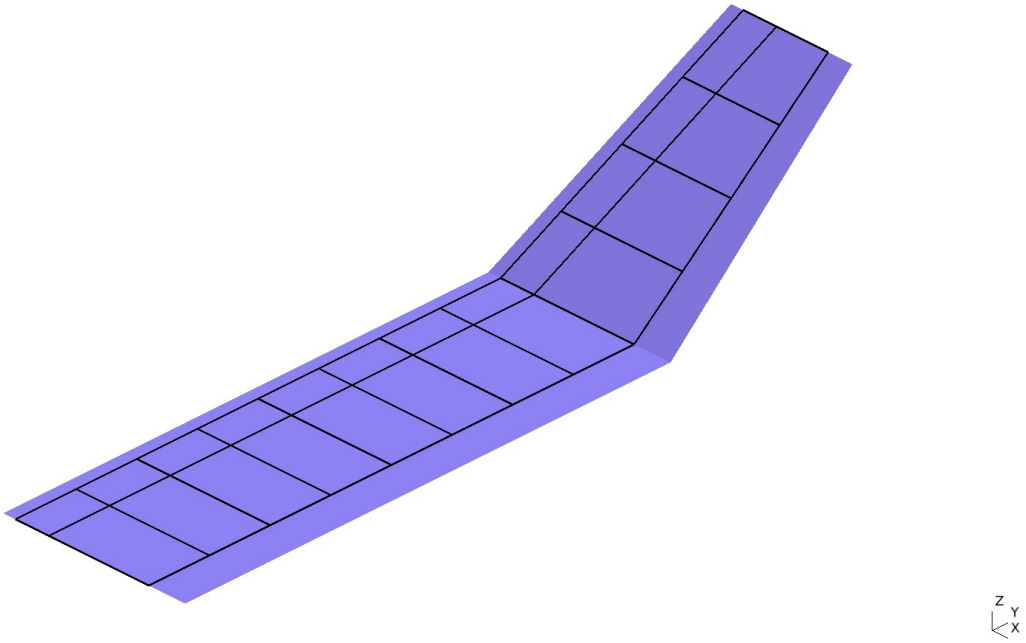


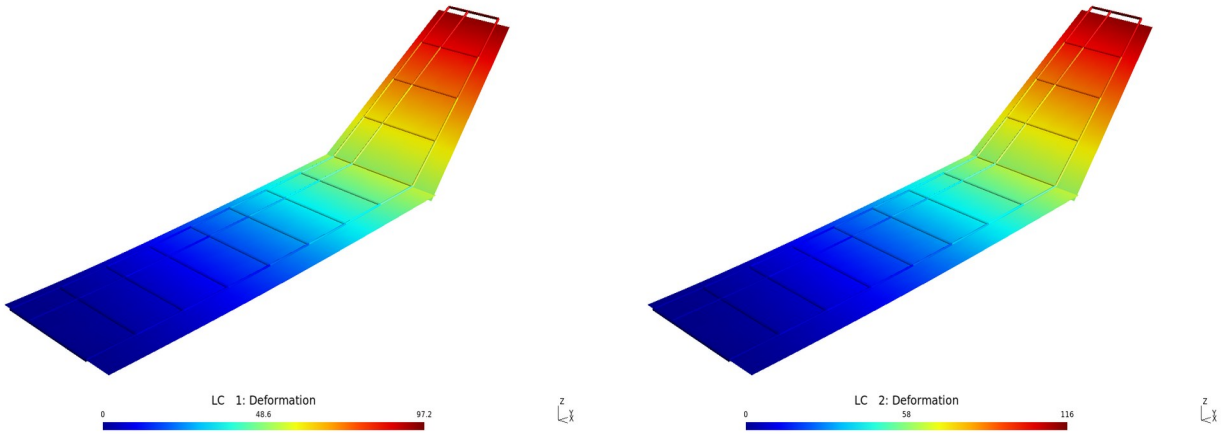
Figure 2.1-8: Gull Wing, Combined Solid and Aerodynamic Models

```
fclose(fid);
```

It is always good practice to look at the combined mesh to check if the geometries of the solid and the aerodynamic mesh are correct. Figure 2.1-8 shows that the geometries are correct.

The quality of the splines can be checked by comparing the displacements of the solid component with those of the aerodynamic component. The displacements normal to the lifting surfaces should match.

Figure 2.1-9 shows a good quality of the splines. When comparing the dis-



Configuration 1 Configuration 2

Figure 2.1-9: Gull Wing, Comparison of Displacements

placements, keep in mind that the aerodynamic mesh only shows displacements normal to the lifting surfaces. This explains why the two components appear to shift relative to each other.

Step 4: Static divergence

From the results of the preceding three steps we are sufficiently confident with the models, so we can start the aeroelastic analysis. First, we compute the dynamic pressure at which static divergence occurs, and the corresponding flight velocity. GNU Octave script `diverg.m` performs these computations:

```
# Example: Gull wing
#
# Static divergence
#
# Files needed: aeroelastic.bin
#
# ------------------------------------------------------------

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

  load aeroelastic.bin

  nofmod = 3;

# Analysis
# --------

  [wings, winga, nfound] = mfs_diverg(wing, nofmod);
  if (nfound)
     mfs_print(fid, winga, "diverg", "qdyn");
     qd = mfs_getresp(winga, "diverg", "qdyn", 1);
     vd = sqrt(2 * qd / rho) / 1000;
     fprintf(fid, "  vd = %7.2f m/s\n", vd);
  endif

  fclose(fid);
```

The dynamic pressure q_D at divergence is the smallest positive real eigenvalue of an unsymmetric eigenvalue problem. As there is no such nice theory for unsymmetric eigenvalue problems as for symmetric ones, we cannot be sure that we obtain this eigenvalue if we compute only one. Thus, we ask Mefisto to compute three eigenvalues. Mefisto stores only the real positive eigenvalues. The total number of eigenvalues found, the number of real eigenvalues found and the number of positive real eigenvalues found is reported:

```
          Eigenvalues of static divergence analysis:
            Found =  3, Real =  3, Positive =  2
   3.8692 seconds needed to perform divergence analysis of component wing
```

The positive real eigenvalues are sorted and written to the output file. The

corresponding velocity can be computed from

$$v_D = \sqrt{\frac{2\,q_D}{\rho}} \quad .$$

The output file contains the following information:

```
------------------------------------------------------------------------

Component "winga"

Dynamic pressure at divergence:

  No.    Dynamic pressure
  ----------------------
   1        6.7273e-02
   2        6.1480e-01

  vd =  333.46 m/s
```

The velocity at which divergence occurs is $333.46\ \mathrm{m/s}$. This is far beyond the $40\ \mathrm{m/s}$ of the analysis. However, it is also far beyond the validity of the incompressible theory. The only conclusion we can draw from this result is that there is no risk of static divergence within the velocity range considered.

Step 5: Aeroelastic analysis of flexible wing

GNU Octave script `flexible.m` repeats the analysis performed in Step 3, but now taking into account the flexibility of the wing. First, the aeroelastic component is loaded. Then, function **`mfs_statresp`** computes the primary aerodynamic results and the displacements of the coupled problem. The input to this function is the structure with the aeroelastic component. The two output arguments are the structures with the solid and the aerodynamic component. These components are subsequently used to compute secondary results and to export the results for postprocessing with Gmsh. We also print the displacements at the leading and trailing edge at the wing tip, i.e. at nodal points 1304 and 1341 (cf. Figure 2.1-3).

```
# Example: Gull wing
#
# Aeroelastic analysis of flexible wing
#
# Files needed: aeroelastic.bin, rigid.plt
#
# ------------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

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

  load aeroelastic.bin
```

```
# Analysis
# --------

# Aeroelastic static response

  [wings, winga] = mfs_statresp(wing);

  mfs_print(fid, wings, "statresp", {"disp", [1301, 1341]});

  winga = mfs_results(winga, "statresp", "panel");
  mfs_export("solid.dsp", "msh", wings, "statresp", "disp");
  mfs_export("aero.prs", "msh", winga,
             "statresp", "pressure", "disp");
```

Next, we compute the load resultants of the aerodynamic component and write them to the output file:

```
# Load resultants

  [FA, MA] = mfs_getresp(winga, "statresp", "aeload");
  nc = columns(FA);

  fprintf(fid, "Load resultants of aerodynamic component: \n\n");
  for k = 1 : nc
      fprintf(fid, "  Configuration %2d:\n", k)
      fprintf(fid, "     F = [%10.3e, %10.3e, %10.3e] kN\n",
              FA(:, k) / 1000);
      fprintf(fid, "     M = [%10.3e, %10.3e, %10.3e] kNm\n",
              MA(:, k) * 1e-6);
  endfor
```

Finally, we retrieve the pressure in the wing sections used in Step 2 and compare it with the results of the rigid wing, obtained from file `rigid.plt`. We create one figure for each of the two configurations. Each figure contains four diagrams corresponding to the four wing sections, and each diagram shows two curves, one for the rigid analysis and one for the flexible analysis.

```
# Comparison of pressure in wing sections

  load rigid.plt

  [xi(:, 1), pfi(:, :, 1), y(1)] = ...
     mfs_xydata(winga, "statresp", "pressure", 1, ycols(1));
  [xi(:, 2), pfi(:, :, 2), y(2)] = ...
     mfs_xydata(winga, "statresp", "pressure", 1, ycols(2));

  [xo(:, 1), pfo(:, :, 1), y(3)] = ...
     mfs_xydata(winga, "statresp", "pressure", [2, 3], ycols(3));
  [xo(:, 2), pfo(:, :, 2), y(4)] = ...
     mfs_xydata(winga, "statresp", "pressure",
                [2, 3], ycols(4));
```

```
pfi = 1000 * pfi; pfo = 1000 * pfo;
xi  = xi / 1000;  xo  = xo / 1000;  y   =  y / 1000;

for k = 1 : length(y)
    header{k} = sprintf("\\eta = %1d\\deg, y = %4.2f m",
                        eta{1}, y(k));
endfor

figure(1, "position", [100, 500, 750, 500],
          "paperposition", [0, 0, 14, 10]);
subplot(2, 2, 1)
   plot(xi(:, 1), squeeze(pri(:, 1, 1)),
        xi(:, 1), squeeze(pfi(:, 1, 1)));
   legend("rigid", "flexible", "location", "northeast");
   title(header{1});
   xlim([0, crm]); ylim([0, 0.9]);
   grid;
   ylabel("p [kPa]");
subplot(2, 2, 2)
   plot(xi(:, 2), squeeze(pri(:, 1, 2)),
        xi(:, 2), squeeze(pfi(:, 1, 2)));
   title(header{2});
   xlim([0, crm]); ylim([0, 0.9]);
   grid;
subplot(2, 2, 3)
   plot(xo(:, 1), squeeze(pro(:, 1, 1)),
        xo(:, 1), squeeze(pfo(:, 1, 1)));
   title(header{3});
   xlim([0, crm]); ylim([0, 0.9]);
```

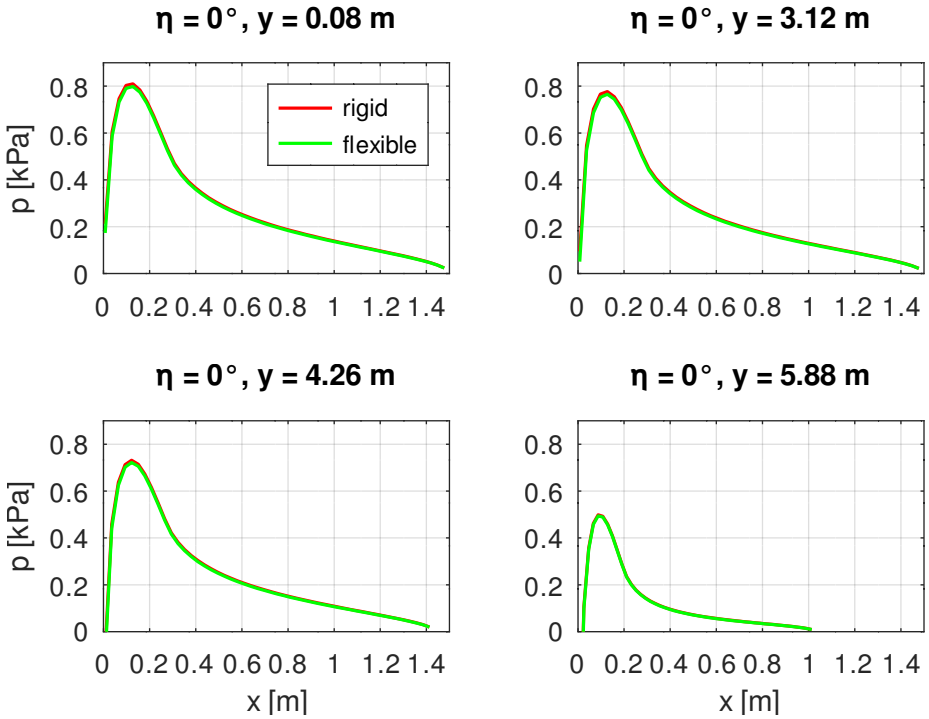


Figure 2.1-10: Gull Wing, Configuration 1, Comparison of Pressure in Wing Sections

```
   grid;
   xlabel("x [m]"); ylabel("p [kPa]");
subplot(2, 2, 4)
   plot(xo(:, 2), squeeze(pro(:, 1, 2)),
        xo(:, 2), squeeze(pfo(:, 1, 2)));
   title(header{4});
   xlim([0, crm]); ylim([0, 0.9]);
   grid;
   xlabel("x [m]");
print(["flexible1", EXT], FORMAT);

for k = 1 : length(y)
    header{k} = sprintf("\\eta = %1d\\deg, y = %4.2f m",
                        eta{2}, y(k));
endfor

figure(2, "position", [250, 250, 750, 500],
          "paperposition", [0, 0, 14, 10]);
subplot(2, 2, 1)
   plot(xi(:, 1), squeeze(pri(:, 2, 1)),
        xi(:, 1), squeeze(pfi(:, 2, 1)));
   legend("rigid", "flexible", "location", "northeast");
   title(header{1});
   xlim([0, crm]); ylim([0, 0.9])
   grid;
   ylabel("p [kPa]");
subplot(2, 2, 2)
   plot(xi(:, 2), squeeze(pri(:, 2, 2)),
        xi(:, 2), squeeze(pfi(:, 2, 2)));
   title(header{2});
```

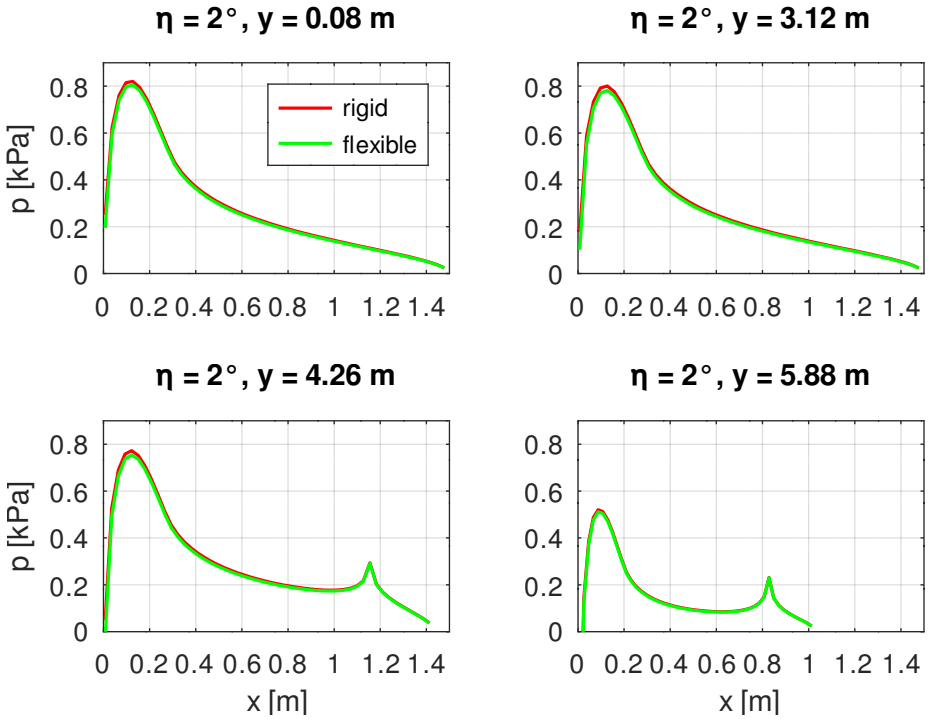


Figure 2.1-11: Gull Wing, Configuration 2, Comparison of Pressure in Wing Sections

```
    xlim([0, crm]); ylim([0, 0.9]);
    grid;
 subplot(2, 2, 3)
    plot(xo(:, 1), squeeze(pro(:, 2, 1)),
         xo(:, 1), squeeze(pfo(:, 2, 1)));
    title(header{3});
    xlim([0, crm]); ylim([0, 0.9]);
    grid;
    xlabel("x [m]"); ylabel("p [kPa]");
 subplot(2, 2, 4)
    plot(xo(:, 2), squeeze(pro(:, 2, 2)),
         xo(:, 2), squeeze(pfo(:, 2, 2)));
    title(header{4});
    xlim([0, crm]); ylim([0, 0.9]);
    grid;
    xlabel("x [m]");
 print(["flexible2", EXT], FORMAT);

 fclose(fid);
```

The output file contains the following results:

```
--------------------------------------------------------------------------------

Component "wings"

Displacements of loadcase  1

   node          ux          uy          uz          rx          ry          rz

    1301    -1.102e+00  -2.009e+01   9.256e+01   2.021e-02  -9.011e-04   3.590e-05
    1341    -1.102e+00  -2.005e+01   9.321e+01   2.020e-02  -9.302e-04   7.454e-05

Displacements of loadcase  2

   node          ux          uy          uz          rx          ry          rz

    1301    -1.898e+00  -2.378e+01   1.085e+02   2.397e-02  -2.061e-03  -7.818e-05
    1341    -1.898e+00  -2.378e+01   1.100e+02   2.394e-02  -2.142e-03   2.159e-05

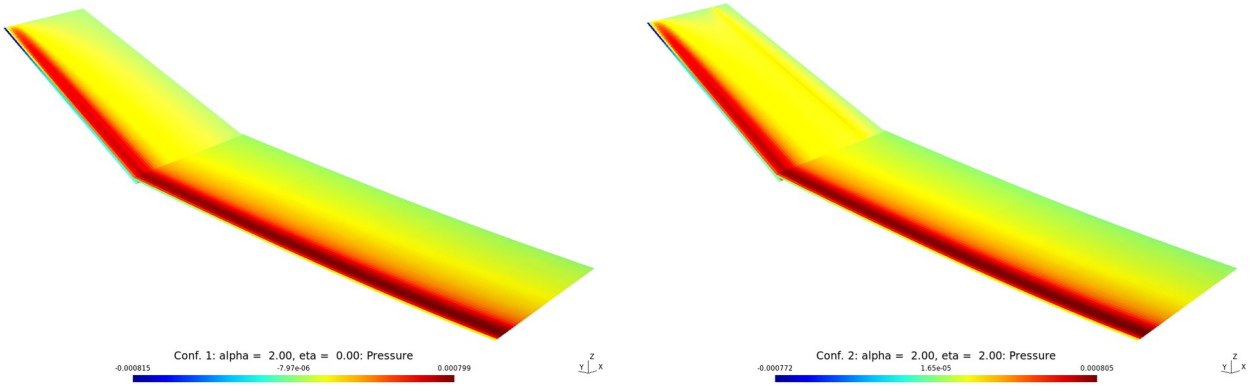

Load resultants of aerodynamic component:

  Configuration  1:
     F = [ 0.000e+00, -2.548e-01,  2.091e+00] kN
     M = [ 5.672e+00, -8.903e-01, -9.734e-02] kNm
  Configuration  2:
     F = [ 0.000e+00, -3.214e-01,  2.270e+00] kN
     M = [ 6.485e+00, -1.014e+00, -1.464e-01] kNm
```

The displacements at the wing tip indicate a small nose down rotation of the wing. Consequently, the resulting lift and moments are slightly smaller than

	Rigid		Flexible	
	F_z [kN]	M_x [kNm]	F_z [kN]	M_x [kNm]
Config. 1	2.136	5.788	2.091	5.672
Config. 2	2.343	6.690	2.270	6.485

Table 2.1-1: Gull Wing, Comparison of Lift and Moment



Configuration 1 Configuration 2

Figure 2.1-12: Gull Wing, Pressure on Deformed Wing [MPa]

for the rigid wing, see Table 2.1-1.

The pressure in the wing sections can be seen in Figures 2.1-10 and 2.1-11. The differences between the rigid and the flexible wing are very small, indicating that the flexibility of the wing has only a very small effect.

Figure 2.1-12 shows the pressure on the deformed wing for both configurations.

2.2 Swept Wing

Summary

Directory:	`exa/aeroelastic/statresp/swept_wing`
Objectives:	• learn how to define a more realistic aeroelastic model • learn how to parametrize your model • learn how to compute aerodynamic coefficients as a function of the dynamic pressure • learn how to obtain stresses in beams
Elements:	**`b2, s4`**
Method:	Vortex-Lattice
Functions:	**`mfs_beamsection, mfs_airfoil, mfs_import, mfs_new, mfs_export, mfs_stiff, mfs_mass, mfs_massproperties, mfs_splines, mfs_freevib, mfs_statresp, mfs_results, mfs_diverg, mfs_print, mfs_beamstress, mfs_getresp, mfs_xydata, mfs_transfer, mfs_merge`**

Problem Description

Perform an aeroelastic analysis of a forward swept wing:

1. Define the solid model and compute the first ten structural modes.
2. Define the aerodynamic model and compute the pressure coefficient, the spanwise lift coefficient and the lift coefficient for the following two configurations:

   a) Angle of attack: 2°; Aileron angle: 0°

   b) Angle of attack: 2°; Aileron angle: 2°

   Also compute the change of the rolling moment due to the aileron deflection.
3. Define the aeroelastic model and check the quality of the splines.
4. Determine the dynamic pressure q_D at which divergence occurs and compute the lift coefficient and the aileron efficiency as functions of q_∞/q_D in the range $0.05 \le q_\infty/q_D \le 0.5$.
5. Perform a detailed analysis of the two configurations at a velocity of $250\ \mathrm{km/h}$. Compute the pressure coefficient in selected wing sections and the spanwise lift coefficient and compare the results with those of the rigid analysis. Determine the maximum stress in the skin and in the stringers.

Figure 2.2-1 shows the plan view of the wing. The wing has the following dimensions: Chord at wing root $c_r = 1200\ \mathrm{mm}$, chord at wing tip $c_t = 800\ \mathrm{mm}$, *y*-coordinate of aileron $y_a = 5000\ \mathrm{mm}$, *y*-coordinate of wing tip $y_t = 8000\ \mathrm{mm}$, sweep angle $\Lambda = -5°$

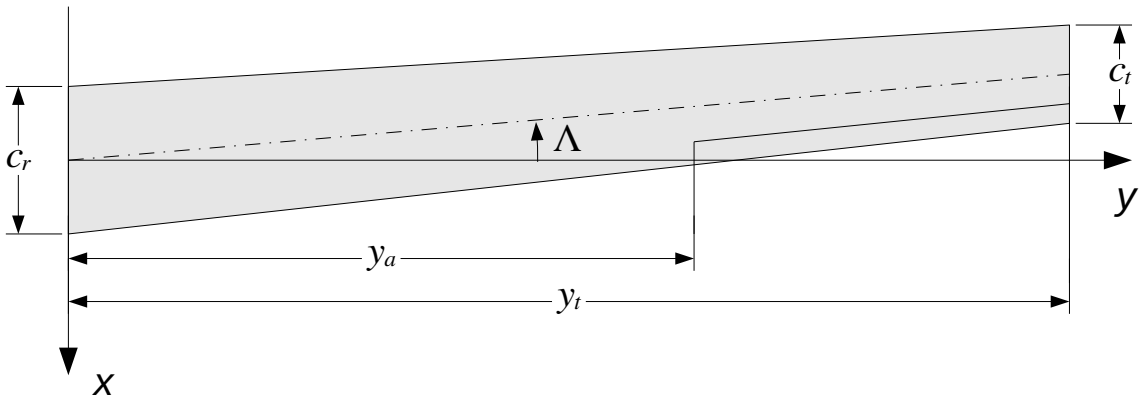


Figure 2.2-1: Plan View of Swept Wing

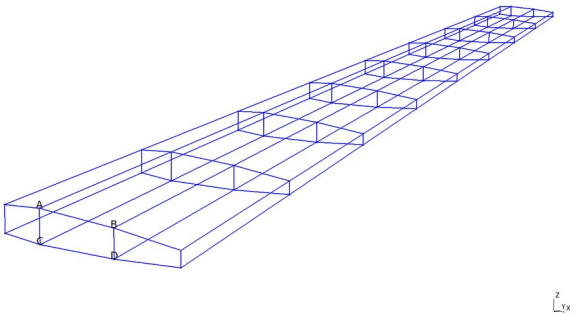


Figure 2.2-2: Geometry of Wing Structure

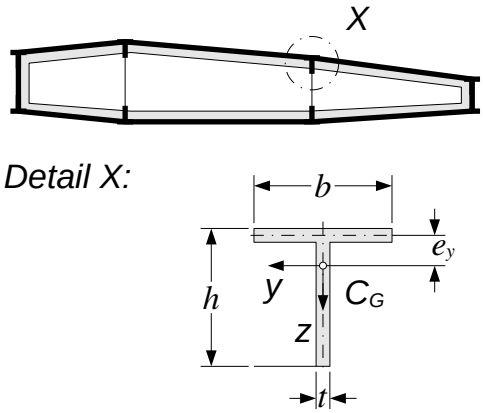


Figure 2.2-3: Wing Section Details

The wing has a NACA 23012 airfoil. The outer wing has a rigging angle of incidence that linearly decreases from 0° at the beginning of the aileron to -3° at the wing tip. The flap chord ratio of the aileron is 20 %.

The solid structure is represented by a box model composed of beam and shell elements (see Figures 2.2-2 and 2.2-3). Its geometry is defined in file `solid.geo`. Table 2.2-1 shows the cross sections and the dimensions of the different physical groups.

The material is aluminium with a Young's modulus of $70000\ \mathrm{MPa}$, a Poisson's ratio of 0.34 and a mass density of $2700\ \mathrm{kg/m^3}$.

The wing is clamped at the wing root. At points A to D (see Figure 2.2-2) all six degrees of freedom are constrained.

Parametrization

All parameters needed to define the solid model, the aerodynamic model and the aeroelastic model are defined in file `params.m`. This file is used by all scripts that need the parameters:

```
# Example: Swept Wing
#
# Definition of all parameters
#
# Units: N, mm, t, degrees
#
# ------------------------------------------------------------

# Geometry
# --------

  sweep =   -5;    % Sweep angle
  ya    = 5000;    % y-position of aileron
```

Physical Group	Type	b mm	h mm	t mm
Ribs_Upper_Part	T-Section	20	20	2
Ribs_Lower_Part	T-Section	20	20	2
Ribs_Front_Part	T-Section	20	20	2
Ribs_End_Part	T-Section	20	20	2
Ribs	Shell			1
Stringer_1_Upper	T-Section	30	30	2
Stringer_2_Upper	T-Section	50	40	4
Stringer_3_Upper	T-Section	40	30	4
Stringer_4_Upper	T-Section	20	20	2
Stringer_1_Lower	T-Section	30	30	2
Stringer_2_Lower	T-Section	50	40	4
Stringer_3_Lower	T-Section	40	30	4
Stringer_4_Lower	T-Section	20	20	2
Front_Spar	Shell			1
Main_Spar	Shell			2
Back_Spar	Shell			2
Rear_Spar	Shell			1
Skin	Shell			1

Table 2.2-1: Cross Sections and Dimensions

```
  yt    = 8000;   % y-position of wing tip
  cr    = 1200;   % Chord at wing root
  ct    =  800;   % Chord at wing tip

# Solid Model
# -----------

# Material: Aluminium

  E   =  70000;
  ny  =   0.34;
  rho = 2.7E-9;

# Cross Sections

  ribs = struct("b", 20, "h", 20, "t", 2);

  stringer = struct("b", {30, 50, 40, 20},
                    "h", {30, 40, 30, 20},
```

```
                    "t", { 2,  4,  4,  2});

# Sheet metal thicknesses

  t_spars = [1, 2, 2, 1];
  t_ribs  = 1;
  t_skin  = 1;

# Aerodynamic Model
# -----------------

# Geometry

  fr   = 0.2;   % Flap chord ratio
  atip =  -3;   % Rigging angle of incidence at wing tip

# Configurations

  alpha =      2; % Angle of attack
  eta   = [0, 2]; % Aileron deflection

# Discretization

  nxi   =   40;   % Panels in x-direction for inner wing
  nxo   =   32;   % Panels in x-direction for outer wing
  nxa   =    8;   % Panels in x-direction for aileron

  nyi   =   15;   % Panels in y-direction for inner wing
  nyo   =   15;   % Panels in y-direction for outer wing

# Wing Section Positions for Pressure Output

  ycoli = [1, floor(0.7 * nyi)]; % Inner wing
  ycolo = [3, floor(0.7 * nyo)]; % Outer wing

# Aeroelastic Model
# -----------------

  nsb   = 10;     % Number of spline breaks
```

The geometrical parameters are also used by file `solid.geo`. If you change these parameters, run script `makeconstants.m` to update the file `constants.geo` which is included by file `solid.geo`.

2.2.1 Solid Model and Normal Modes

GNU Octave script `solid.m` defines the solid model and computes the first ten normal modes.

Model Definition

The script begins with the definition of the translation data.

```
# Example: Swept Wing
#
# Solid model:
#  - Definition of the solid model
#  - Computation of first 10 normal modes
#  - Storage of solid component in binary file solid.bin
#  - Storage of translation data in binary data eset.bin
#
# ------------------------------------------------------------------

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

  nofmod = 10;      % Number of normal modes

# Model Definition
# ----------------

# Parameters

  params

# Translation data

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

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

# Ribs

  [geom, e] = mfs_beamsection("T", ribs.b, ribs.h, ribs.t);
  geom.v    = [0, 0, -1];
  geom.P    = [0, -e(1)];
  Ribs_Upper_Part = struct("type", "elements", "name", "b2",
                           "geom", geom, "mat", mat);

  Ribs_Lower_Part        = Ribs_Upper_Part;
  Ribs_Lower_Part.geom.v = [0, 0, 1];

  data.Ribs_Upper_Part = Ribs_Upper_Part;
  data.Ribs_Lower_Part = Ribs_Lower_Part;

  Ribs_Front_Part = Ribs_Upper_Part;
  Ribs_Front_Part.geom.v = [1, 0, 0];

  Ribs_End_Part = Ribs_Front_Part;
  Ribs_End_Part.geom.v = [-1, 0, 0];

  data.Ribs_Front_Part = Ribs_Front_Part;
  data.Ribs_End_Part   = Ribs_End_Part;

  geoms = struct("t", t_ribs);

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

```
# Stringers

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

  for n = 1 : 4

      [geom, e] = mfs_beamsection("T", stringer(n).b,
                                       stringer(n).h,
                                       stringer(n).t);
      geom.v = [0, 0, -1];
      geom.P = [0, -e(1)];
      Stringer.geom = geom;

      Upper = sprintf("Stringer_%d_Upper", n);
      Lower = sprintf("Stringer_%d_Lower", n);

      data.(Upper) = Stringer;
      Stringer.geom.v = [0, 0, 1];
      data.(Lower) = Stringer;

  endfor

# Spars

  Spar_Names = {"Front_Spar", "Main_Spar", "Back_Spar", ...
                "Rear_Spar"};

  Spar = struct("type", "elements", "name", "s4",
                "mat", mat);

  for n = 1 : 4
      Spar.geom = struct("t", t_spars(n));
      data.(Spar_Names{n}) = Spar;
  endfor

# Skin

  geoms = struct("t", t_skin);

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

# Constraints

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

Analysis

First, we use function **`mfs_import`** to create a Mefisto model description from the mesh file and the translation data. The model description is returned

in variable **model**.

Next, the solid component is created, and the beam axes are exported to Gmsh for checking their correct definition.

```
# Analysis
# --------

  model = mfs_import(fid, "solid.msh", "msh", data);
  wings = mfs_new(fid, model);
  mfs_export("solid.axes", "msh", wings, "mesh", "axes");
```

Subsequently, the stiffness and the mass matrix are computed, and the component and the translation data are saved to binary files `solid.bin` and `data.bin` respectively.

```
  wings = mfs_stiff(wings);
  wings = mfs_mass(wings);
  mfs_massproperties(fid, wings);

  save -binary solid.bin wings
  save -binary data.bin data
```

Finally, the normal modes are computed and exported to Gmsh for post-processing.

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

  fclose(fid);
```

The output file contains the following information:

```
Reading model from file "solid.msh", MSH file version 4.1

  Physical Group                    Type
  ------------------------------------------------
  Ribs_Upper_Part                   elements
  Ribs_Lower_Part                   elements
  Ribs_Front_Part                   elements
  Ribs_End_Part                     elements
  Stringer_1_Upper                  elements
  Stringer_2_Upper                  elements
  Stringer_3_Upper                  elements
  Stringer_4_Upper                  elements
  Stringer_1_Lower                  elements
  Stringer_2_Lower                  elements
  Stringer_3_Lower                  elements
  Stringer_4_Lower                  elements
  Front_Spar                        elements
  Main_Spar                         elements
  Back_Spar                         elements
  Rear_Spar                         elements
  Ribs                              elements
  Skin                              elements
  Clamped                           constraints

Mefisto 2.8: Building new component from input "model"
```

```
  Model Type = solid, Model Subtype = 3d

  Number of nodes      =   264,  Number of elements =   675
  Number of element types =  2
  Number of global     degrees of freedom =   1584
  Number of local      degrees of freedom =   1560
  Number of prescribed degrees of freedom =     24
  Number of dependent  degrees of freedom =      0


Mass properties of component "wings"

  Coordinates of reference point:      0.0000,     0.0000,      0.0000

  Rigid body mass matrix:

     8.2217e-02 -3.3205e-19  9.9326e-21  3.9995e-17  1.0644e+00 -3.1536e+02
    -3.2748e-19  8.2217e-02 -1.3453e-22 -1.0644e+00 -1.0501e-17 -1.4474e+01
     8.1564e-21 -2.5648e-21  8.2217e-02  3.1536e+02  1.4474e+01 -2.5444e-17
     3.5915e-17 -1.0644e+00  3.1536e+02  1.6531e+06  9.8081e+04  2.4508e+02
     1.0644e+00 -1.1687e-17  1.4474e+01  9.8081e+04  1.1022e+04 -3.8039e+03
    -3.1536e+02 -1.4474e+01 -2.9462e-17  2.4508e+02 -3.8039e+03  1.6637e+06

  Mass =  8.2217e-02

  Inertia tensor with respect to reference point:

     1.6531e+06  9.8081e+04  2.4508e+02
     9.8081e+04  1.1022e+04 -3.8039e+03
     2.4508e+02 -3.8039e+03  1.6637e+06

  Coordinates of center of mass:  -176.0441,  3835.6957,    12.9466

  Inertia tensor with respect to center of mass:

     4.4346e+05  4.2564e+04  5.7693e+01
```

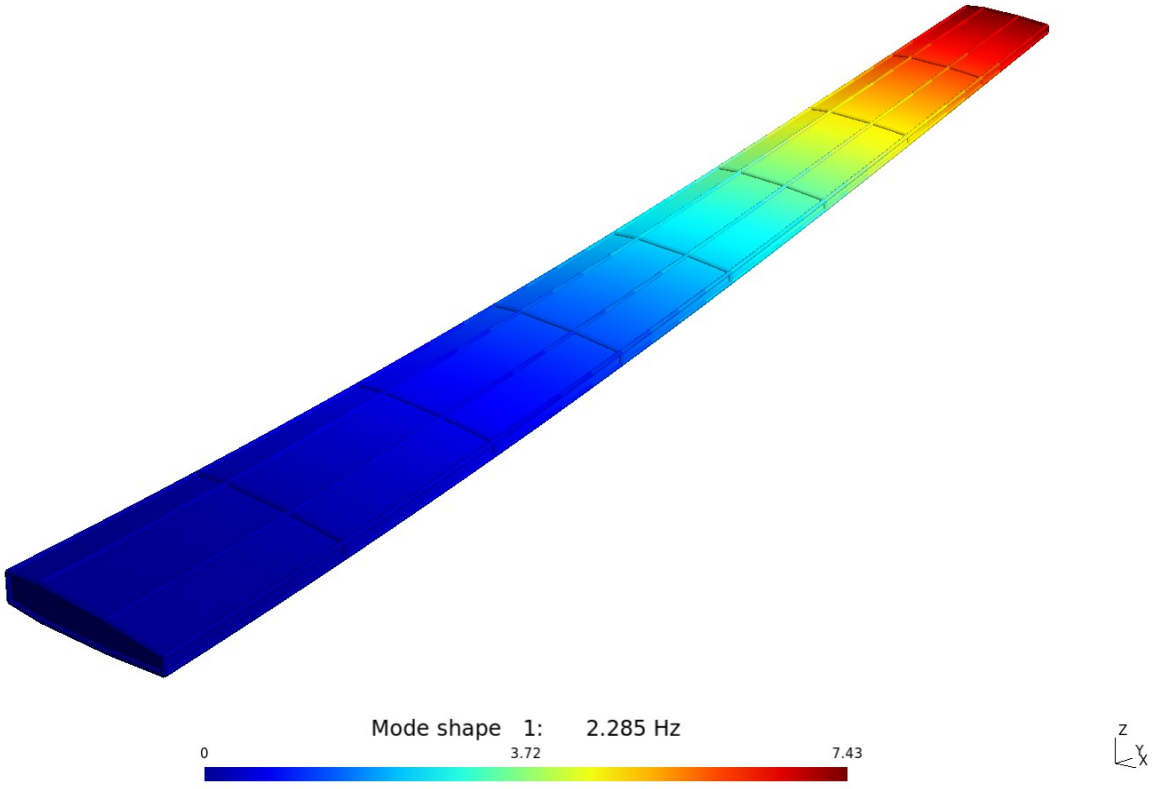


Figure 2.2-4: Swept Wing, First Bending Mode

```
     4.2564e+04  8.4602e+03  2.7890e+02
     5.7693e+01  2.7890e+02  4.5153e+05

-------------------------------------------------------------------------------

Component "wings"

Natural frequencies:

  Mode    Circ. Frequency    Frequency
  ------------------------------------
     1         14.35523        2.28471 Hz
     2         58.46222        9.30455 Hz
     3         77.88258       12.39540 Hz
     4        191.18140       30.42746 Hz
     5        206.53902       32.87171 Hz
     6        307.41273       48.92626 Hz
     7        392.62458       62.48814 Hz
     8        478.61948       76.17466 Hz
     9        631.19868      100.45839 Hz
    10        774.31013      123.23529 Hz
```

Figure 2.2-4 shows the first bending mode and Figure 2.2-5 the first torsion mode.

2.2.2 Aerodynamic Model and Aerodynamic Coefficients

GNU Octave script `rigid.m` defines the aerodynamic model and computes the pressure and lift coefficients for the two configurations.

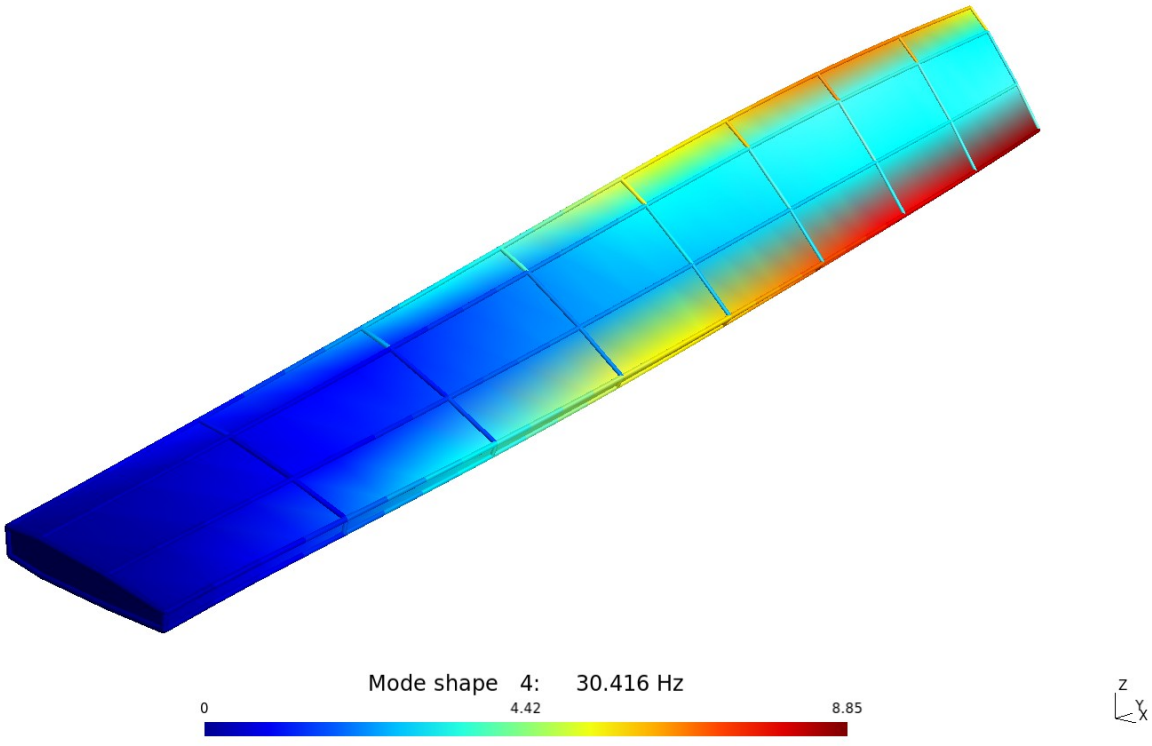


Figure 2.2-5: Swept Wing, First Torsion Mode

Model Definition

The definition of the aerodynamic model is similar to Example 2.1. However, we do not define a value of the dynamic pressure so that the default value of $q_\infty = 1$ is used. Thus, the aerodynamic pressure will actually be a pressure coefficient, and forces and moments will be forces and moments divided by dynamic pressure.

```
# Example: Swept Wing
#
# Aerodynamic analysis of rigid wing
#  - Definition of the aerodynamic model
#  - Computation of pressure coefficient, lift coefficient and
#    increase of rolling moment due to aileron deflection
#  - Storage of aerodynamic model and component in binary file
#    aero.bin
#  - Storage of xy-plot data in binary file rigid.plt
#
# ------------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

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

# Model Definition
# ----------------

# Parameters

  params
  frc = 1 - fr;

# Airfoil data

  cmbi = mfs_airfoil("NACA", 5, 30);
  cmbo = mfs_airfoil("NACA", 5, 30, 0, 1 - fcr);
  cmba = mfs_airfoil("NACA", 5, 30, 1 - fcr, 1);

# Model type, subtype and symmetry

  aero = struct("type", "aero", "subtype", "vlm", "symy", 0);

# Points on leading edge of wing

  sa = ya / yt;
  ca = (1 - sa) * cr + sa * ct;
  sw = tand(sweep);

  xr = -0.25 * cr;
  xa = ya * sw - 0.25 * ca;
  xt = yt * sw - 0.25 * ct;
```

```
  points(1 : 3) = struct("id",   {1, 2, 3},
                         "coor", {[xr,  0, 0], [xa, ya, 0], ...
                                  [xt, yt, 0]});

# Points on leading edge of aileron

  xa += frc * ca; xt += frc * ct;

  points(4 : 5) = struct("id",   {4, 5},
                         "coor", {[xa, ya, 0], [xt, yt, 0]});

# Lifting surfaces of inner wing, outer wing and aileron

  lsf = struct("id",     {1, 2, 3},
               "points", {[1, 2], [2, 3], [4, 5]},
               "chord",  {[cr, ca], frc * [ca, ct], ...
                          fr * [ca, ct]},
               "camber", {cmbi, cmbo, cmba},
               "nx",     {nxi, nxo, nxa},
               "ny",     {nyi, nyo, nyo},
               "typex",  "linear",
               "typey",  {"linear", "cos>", "cos>"},
               "alpha",  {0, [0, atip], [0, atip]});

# Aileron is a control surface

  controls = struct("name", "aileron", "ls", 3);

# Configurations

  for n = 1 : 2
      cfgname = sprintf("Conf. %d: alpha = %5.2f, eta = %5.2f",
                        n, alpha, eta(n));
      config(n) = struct("name", cfgname,
                         "alpha", alpha, "aileron", eta(n));
  endfor

# Add definitions to structure aero

  aero.points   = points;
  aero.ls       = lsf;
  aero.controls = controls;
  aero.config   = config;
```

Analysis

The analysis is very similar to that of Example 2.1, except that not only the aerodynamic component but also the model definition are saved. This is necessary because in subsequent steps of the analysis, new configurations will be defined.

Because the dynamic pressure used in the analysis is unity, the pressures

are already pressure coefficients, and the forces and moments are already forces and moments divided by the dynamic pressure, so they only need to be divided by the wing area.

```
# Analysis
# --------

  winga = mfs_new(fid, aero);
  mfs_export("aero.msh", "msh", winga, "mesh", "camber");

  winga = mfs_statresp(winga);
  winga = mfs_results(winga, "statresp", "panel");

  mfs_export("rigid.pos", "msh", winga, "statresp", "pressure");

# Lift coefficient

  [F, M] = mfs_getresp(winga, "statresp", "aeload");
  A      = mfs_getresp(winga, "mesh", "area");
  cLR    = F(3, 1) / A;
  dMxR   = M(1, 2) - M(1, 1);

  fprintf(fid, "Area = %11.4e\n", A);
  fprintf(fid, "cLR  = %11.4e, dMxR = %11.4e\n", cLR, dMxR);

  save -binary aero.bin aero winga

# Pressure coefficient in selected wing sections

  [xi(:, 1), pri(:, :, 1), y(1)] = ...
     mfs_xydata(winga, "statresp", "pressure", 1, ycoli(1));
  [xi(:, 2), pri(:, :, 2), y(2)] = ...
     mfs_xydata(winga, "statresp", "pressure", 1, ycoli(2));
  [xo(:, 1), pro(:, :, 1), y(3)] = ...
     mfs_xydata(winga, "statresp", "pressure", [2, 3], ycolo(1));
  [xo(:, 2), pro(:, :, 2), y(4)] = ...
     mfs_xydata(winga, "statresp", "pressure", [2, 3], ycolo(2));

  xi = xi / cr; xo = xo / cr;
  y  =  y * 1e-3;  % convert to m

  for n = 1 : length(y)
      leg{n} = sprintf("y = %4.2f m", y(n));
  endfor
  for n = 1 : length(eta)
      header{n} = sprintf("\\eta = %2.0f\\deg", eta(n));
  endfor

  figure(1, "position", [100, 500, 1000, 500],
            "paperposition", [0, 0, 16, 10]);
  subplot(1, 2, 1)
     plot(xi(:, 1), squeeze(pri(:, 1, 1)),
          xi(:, 2), squeeze(pri(:, 1, 2)),
          xo(:, 1), squeeze(pro(:, 1, 1)),
          xo(:, 2), squeeze(pro(:, 1, 2)));
```

```
    legend(leg, "location", "northeast");
    title(header{1});
    ylim([-0.1, 1]);
    grid;
    xlabel('x / c'); ylabel('\Delta c_P');
 subplot(1, 2, 2)
    plot(xi(:, 1), squeeze(pri(:, 2, 1)),
         xi(:, 2), squeeze(pri(:, 2, 2)),
         xo(:, 1), squeeze(pro(:, 2, 1)),
         xo(:, 2), squeeze(pro(:, 2, 2)));
    title(header{2});
    ylim([-0.1, 1]);
    grid;
    xlabel('x / c');
 print(["rigid", EXT], FORMAT);
```

In addition to the pressure coefficient we also plot the spanwise lift coefficient

$$c_l = \frac{1}{c_r q_\infty} \frac{dL}{ds} \ .$$

where s is the coordinate in spanwise direction and c_r is the chord at the wing root. Because of $q_\infty = 1$ the force per length obtained with **`mfs_xydata`** must only be divided by the reference length.

```
# Loads in spanwise direction

  [s, f, m] = mfs_xydata(winga, "statresp", "spanload",
```

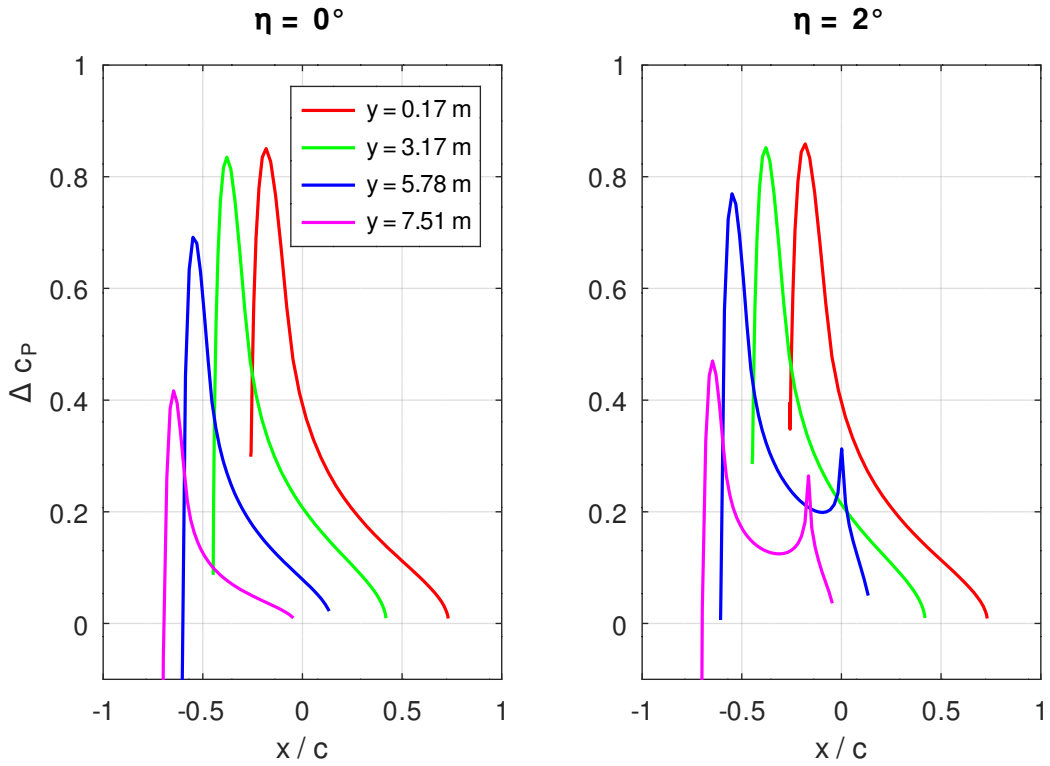


Figure 2.2-6: Pressure Coefficient of Rigid Wing

```
                          {1, [2, 3]});
  cl = squeeze(f(3, :, :)) / cr;

  figure(2, "position", [100, 500, 760, 500],
            "paperposition", [0, 0, 15, 10]);
  plot(s * 1e-3, cl);
  legend(header)
  grid;
  xlabel('s [m]'); ylabel('c_l');
  print(["rigid_cl", EXT], FORMAT);
```

Finally, some of the results are stored in structure **rigres** which is saved to a binary file so that they can be compared with results from the aeroelastic analysis.

```
# Save results for script stress.m

  rigres = struct("cL", cLR, "dMx", dMxR, "pi", pri, "po", pro,
                  "cl", cl);
  save -binary rigres.bin rigres

  fclose(fid);
```

The output file contains the following information:

```
Mefisto 2.8: Building new component from input "aero"

  Model Type = aero, Model Subtype = vlm

  Number of nodes  =  1328,  Number of panels =  1200
```

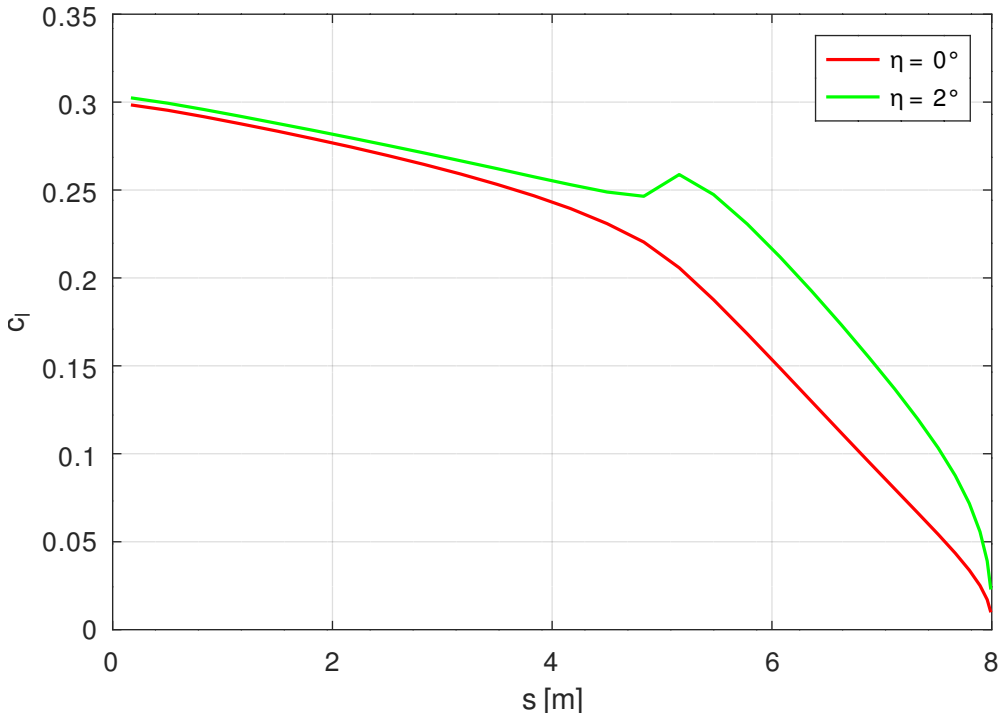


Figure 2.2-7: Spanwise Lift Coefficient of Rigid Wing

```
  Number of lifting surfaces =            3
  Number of control surfaces =            1
  Number of configurations   =            2
  Reference chord length     =  0.00000e+00
  Symmetry plane: y          =  0.00000e+00

Area =  8.0000e+06
cLR  =  2.5364e-01, dMxR =  1.4552e+09
```

Figure 2.2-6 shows the pressure coefficient in selected wing sections for the two configurations, and Figure 2.2-7 shows the spanwise lift coefficient.

2.2.3 Aeroelastic Model

GNU Octave script `splines.m` defines the splines and checks their quality by transferring the aerodynamic loads to the solid component, comparing the load resultants and computing the displacements.

Model Definition

The definition of the aeroelastic model begins with the definition of the model type and the solid and aerodynamic components:

```
# Example: Swept Wing
#
# Splines:
#  - Definition of the aeroelastic model
#  - Transfer of aerodynamic loads to solid model
#  - Computation of displacements
#  - Storage of aeroelastic model and component in binary file
#    aeroelastic.bin
#
# Files needed: solid.bin, aero.bin
#
# ------------------------------------------------------------

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

# Aeroelastic model
# -----------------

  params

  load solid.bin
  load aero.bin

  model = struct("type", "aeroelastic",
                 "solid", wings, "aero", winga);
```

Next, the splines are defined. In contrast to Example 2.1 all lifting surfaces have the same normal vector. We therefore have to define only one spline that is connected to all nodal points. As this is the default, no nodal points

sets need be defined.

```
# Splines

  data = struct("nbreaks", nsb);

  model.splines = struct("id", 1,        "type", "tb",
                         "lsid", 1 : 3, "data", data);
```

Analysis

The analysis steps are very similar to those of Example 2.1. However, in addition to the aeroelastic component, we also save the aeroelastic model. Then, it is easy to create a new aeroelastic component if the aerodynamic component is changed.

```
# Create component and compute splines

  wing = mfs_new(fid, model);
  wing = mfs_splines(wing);

  save -binary aeroelastic.bin model wing

# Transfer aerodynamic loads from rigid aerodynamic analysis
# to the solid component and retrieve load resultants

  wings = mfs_transfer(wing, winga, "statresp", "loads");
  [FA, MA] = mfs_getresp(winga, "statresp", "aeload");
  [FS, MS] = mfs_getresp(wings, "load", "resultant");

  nc = columns(FA);
```

The splines are checked by comparing the lift and moment coefficients obtained from the loads on the aerodynamic and the solid model. The reference length for the moment coefficient about the *x*-axis is the length of the wing.

```
# Compute aerodynamic coefficients

  A = mfs_getresp(winga, "mesh", "area");

  cLA = FA(3, :) / A;
  cMA = MA(1, :) / (yt * A);
  cLS = FS(3, :) / A;
  cMS = MS(1, :) / (yt * A);

  fprintf(fid, "Aerodynamic coefficients:\n\n");
  fprintf(fid, "Config.   cLA     cLS    cMxA    cMxS  \n");

  for n = 1 : nc
      fprintf(fid, "  %2.0d    %6.4f  %6.4f  %6.4f  %6.4f\n",
              n, cLA(n), cLS(n), cMA(n), cMS(n));
```

```
  endfor
```

The following information is written to the output file:

```
Mefisto 2.8: Building new component from input "model"

  Model Type = aeroelastic

  Number of splines      =   1

Aerodynamic coefficients:

Config.    cLA      cLS     cMxA     cMxS
   1     0.2536   0.2536   0.1004   0.1004
   2     0.2850   0.2850   0.1231   0.1231
```

It can be seen that the aerodynamic coefficients computed from the pressure coefficient are identical with those computed from the loads on the solid structure.

Finally, displacements are computed and transferred to the aerodynamic mesh, and the meshes and the results are merged. The displacements correspond to a dynamic pressure of $1\ \mathrm{MPa}$. Thus, their absolute values are very large. However, we use these displacements only to check the quality of the splines.

```
# Compute displacements and transfer them to the aerodynamic
```

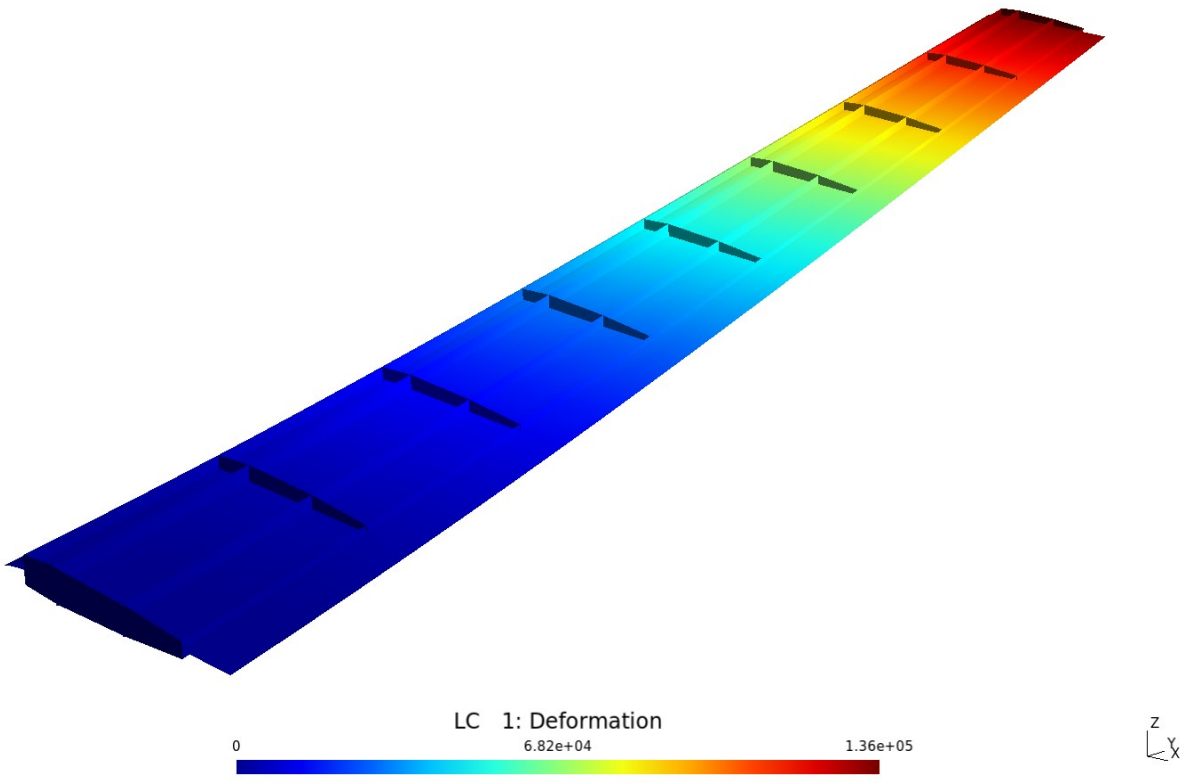


Figure 2.2-8: Comparison of Solid and Aerodynamic Displacements

```
# component

  wings = mfs_statresp(wings);
  winga = mfs_transfer(wing, wings, "statresp", "disp");

  mfs_export("solid.lds", "msh", wings, "load", "point");
  mfs_export("solid.dsp", "msh", wings, "statresp", "disp");
  mfs_export("aero.dsp", "msh", winga, "statresp", "disp");

# Combine the models and the displacements

  mfs_merge("solid.msh", "aero.msh", "wing.msh", "msh");
  mfs_merge("solid.dsp", "aero.dsp", "wing.dsp", "msh");

  fclose(fid);
```

Figure 2.2-8 shows the displacements of the combined solid and aerodynamic meshes. For better visibility, the skin of the solid structure is not plotted. Figure 2.2-8 shows that the displacements of the aerodynamic mesh are very smooth and closely follow those of the solid mesh.

2.2.4 Static Divergence, Lift Coefficient and Aileron Efficiency

GNU Octave script `flexible.m` uses the components defined in the previous steps to perform some aeroelastic analyses. First, a static divergence analysis is performed. Subsequently, the lift coefficient and the aileron efficiency are computed as functions of the dynamic pressure.

Static Divergence

The aeroelastic component is loaded. Subsequently, function **`mfs_diverg`** is used to compute five eigenvalues. The positive real eigenvalues corresponding to dynamic pressures at which divergence occurs are written to the output file.

```
# Example: Swept Wing
#
# Aeroelastic analysis of flexible wing
#
# Files needed: aero.bin, aeroelastic.bin, rigres.bin
#
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10)

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

  params
```

```
# Static divergence
# -----------------

  load aeroelastic.bin

  [wings, winga, nfound] = mfs_diverg(wing, 5);

  if (nfound)
     mfs_print(fid, winga, "diverg", "qdyn");
     qd = mfs_getresp(winga, "diverg", "qdyn", 1);
  else
     fclose(fid);
     return;
  endif
```

This part of the script writes the following information to the output file:

```
Component "winga"

Dynamic pressure at divergence:

  No.    Dynamic pressure
  ----------------------
   1        1.1804e-02
   2        1.6547e-01
   3        6.3606e-01
   4        1.5021e+00
   5        2.8531e+00
```

Static divergence first occurs at a dynamic pressure of $1.180 \cdot 10^{-2}$ MPa. With a mass density of the air of 1.21 kg/m^3 this corresponds to a velocity of

$$v_D = \sqrt{\frac{2 \cdot 1.180 \cdot 10^{-2} \,\mathrm{N/mm^2}}{1.21 \cdot 10^{-12} \,\mathrm{Ns^2/mm^4}}} = 140 \,\frac{\mathrm{m}}{\mathrm{s}} \ .$$

This is about 41 % of the speed of sound, i.e. this prediction is beyond the limit of validity of the incompressible theory.

Lift Coefficient and Aileron Efficiency

The lift coefficient and the aileron efficiency are computed for different values of the dynamic pressure in the range from $0.05 q_D$ to $0.5 q_D$. The upper value corresponds to a velocity of

$$v_\infty = \sqrt{0.5}\, v_D = 99.0 \,\mathrm{m/s} \ .$$

This is less than about 30 % of the speed of sound so that the air can be considered incompressible.

For each dynamic pressure two configurations are defined. The angle of attack of both configurations is 2°. The first **nq** configurations correspond to an aileron deflection angle of 0° and the last **nq** to an aileron deflection of 2°, **nq** being the number of dynamic pressure values.

```
# Lift coefficient and aileron efficiency
# ---------------------------------------

  load aero.bin
  load rigres.bin

# Define new configurations

  qr   = [0.05 : 0.05 : 0.5];
  nq   = length(qr);
  qdyn = qr * qd;

  nc = 1;

  for m = 1 : 2
      for n = 1 : nq
          name = sprintf("Conf. %d: qdyn = %10.4e, eta = %5.2f",
                         nc, qdyn(n), eta(m));
          config(nc++) = struct("name",    name,
                                "qdyn",    qdyn(n),
                                "alpha",   alpha,
                                "aileron", eta(m));
      endfor
  endfor

  aero.config = config;
```

Next, a new aerodynamic component is created and assigned to field **aero** of structure **model** that describes the aeroelastic model. Then, a new aeroelastic component is created.

```
# Create new aerodynamic component

  winga = mfs_new(fid, aero);

# Create new aeroelastic component

  model.aero = winga;

  wing = mfs_new(fid, model);
```

Subsequently, the splines and the aeroelastic results are computed. The aerodynamic results are stored in the aerodynamic component (structure **winga**) and the structural results are stored in the solid component (structure **wings**).

```
# Compute splines and results

  wing = mfs_splines(wing);

  [wings, winga] = mfs_statresp(wing);
  winga = mfs_results(winga, "statresp", "panel");
```

Finally, the lift coefficient and the aileron efficiency can be computed from the load resultants. The aileron efficiency is defined as

$$\eta_R = \frac{\Delta M_x^F}{\Delta M_x^R}$$

where the superscript F denotes the flexible and the superscript R the rigid increment of the rolling moment due to the aileron deflection.

```
# Compute lift coefficient

  A      = mfs_getresp(winga, "mesh", "area");
  [F, M] = mfs_getresp(winga, "statresp", "aeload");

  fprintf(fid,
          "     qdyn          Fx           Fy           Fz      ");
  fprintf(fid, "    Mx           My           Mz\n");
  for n = 1 : nq
      fprintf(fid, "  %10.4e  %10.4e  %10.4e  %10.4e",
              qdyn(n), F(:, n));
      fprintf(fid, "  %10.4e  %10.4e  %10.4e\n", M(:, n));
  endfor

  cLF = F(3, 1 : nq) ./ (A * qdyn);
  cLr = cLF / rigres.cL;

# Compute aileron efficiency

  dMxF = M(1, nq + 1 : end) - M(1, 1 : nq);
  dMxF = dMxF ./ qdyn;
  etaR = dMxF / rigres.dMx;

# Plot results

  figure(1, "position", [100, 500, 1000, 500],
            "paperposition", [0, 0, 17, 10]);
  subplot(1, 2, 1);
     plot(qr, cLr);
     grid;
     xlabel('q_\infty / q_D'); ylabel("c_L/c_{LR}");
  subplot(1, 2, 2);
     plot(qr, etaR);
     grid;
     xlabel('q_\infty / q_D'); ylabel('\eta_R');
  print(["flexible", EXT], FORMAT);

  fclose(fid);
```

The load resultants of the first **nq** configurations, i.e. the configurations without aileron deflection, are written to the output file:

```
Mefisto 2.7: Building new component from input "model"
```

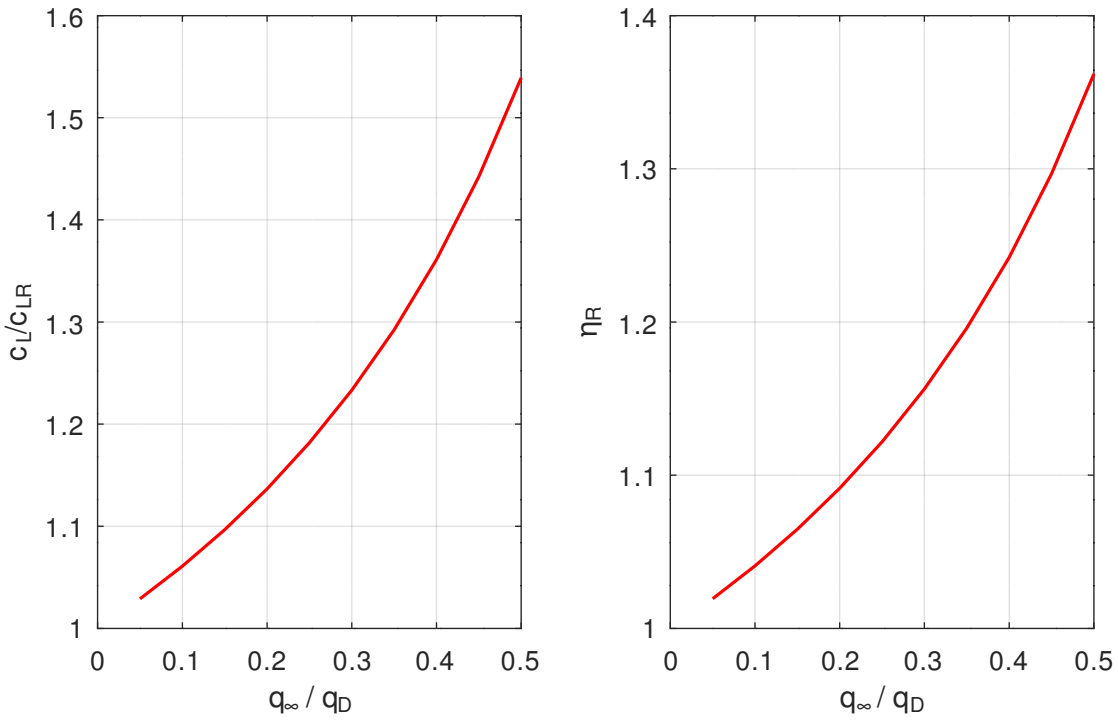


Figure 2.2-9: Lift Coefficient and Aileron Efficiency

```
Model Type = aeroelastic

Number of splines      =   1

   qdyn          Fx          Fy          Fz          Mx          My          Mz
5.9022e-04  0.0000e+00  0.0000e+00  1.2323e+03  3.9357e+06  2.8647e+05  0.0000e+00
1.1804e-03  0.0000e+00  0.0000e+00  2.5413e+03  8.1925e+06  6.0127e+05  0.0000e+00
1.7707e-03  0.0000e+00  0.0000e+00  3.9402e+03  1.2827e+07  9.4935e+05  0.0000e+00
2.3609e-03  0.0000e+00  0.0000e+00  5.4450e+03  1.7908e+07  1.3369e+06  0.0000e+00
2.9511e-03  0.0000e+00  0.0000e+00  7.0766e+03  2.3526e+07  1.7718e+06  0.0000e+00
3.5413e-03  0.0000e+00  0.0000e+00  8.8614e+03  2.9795e+07  2.2641e+06  0.0000e+00
4.1316e-03  0.0000e+00  0.0000e+00  1.0834e+04  3.6863e+07  2.8270e+06  0.0000e+00
4.7218e-03  0.0000e+00  0.0000e+00  1.3041e+04  4.4931e+07  3.4781e+06  0.0000e+00
5.3120e-03  0.0000e+00  0.0000e+00  1.5544e+04  5.4271e+07  4.2413e+06  0.0000e+00
5.9022e-03  0.0000e+00  0.0000e+00  1.8433e+04  6.5261e+07  5.1502e+06  0.0000e+00
```

Figure 2.2-9 shows the lift coefficient and the aileron efficiency. The lift coefficient is scaled by the lift coefficient of the rigid wing which does not depend on the dynamic pressure. It can be seen that the lift coefficient increases with increasing dynamic pressure. This indicates that the wing sections rotate nose up. This rotation is due to the bending. Also, the aileron efficiency increases, indicating that the nose down effect of the additional torsional moment caused by the aileron deflection is more than compensated by the nose up effect due to bending. This is a typical behaviour of forward swept wings.

2.2.5 Detailed Analysis at 250 km/h

Finally, GNU Octave script `stress.m` performs a detailed analysis of the flexible wing at a velocity of 250 km/h. Apart from the value of the dynamic pressure, the configurations considered are identical to those studied in case

of the rigid wing (Section 2.2.2).

First, we define the new configurations and create the aerodynamic and aeroelastic components.

```
# Example: Swept Wing
#
# Stress analysis at a velocity of 250 km/h
#
# Files needed: aero.bin, aeroelastic.bin
#
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

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

# Data

  params

  v   =      250;    % Flight velocity in km/h
  rho = 1.21e-12;    % Mass density of air in t/mm^3

# Define new configurations

  load aero.bin

  vmms = 1000 * v / 3.6;
  qdyn = 0.5 * rho * vmms^2;

  for n = 1 : 2
      name{n} = sprintf("Conf. %d: qdyn = %10.4e, eta = %5.2f",
                        n, qdyn, eta(n));
  endfor
  aero.config = struct("name", name, "qdyn", qdyn, "alpha", alpha,
                       "aileron", num2cell(eta));

# Create new aerodynamic component

  winga = mfs_new(fid, aero);

# Create new aeroelastic component

  load aeroelastic.bin

  model.aero = winga;

  wing = mfs_new(fid, model);
```

Next, we compute the splines, the pressure, the displacements, the stresses and the stress resultants and export these results for post-processing with

Gmsh.

```
# Compute and export results

  wing = mfs_splines(wing);

  [wings, winga] = mfs_statresp(wing);
  winga = mfs_results(winga, "statresp", "panel");
  wings = mfs_results(wings, "statresp", "element");

  mfs_export("aero.pos", "msh", winga, "statresp",
             "pressure", "disp");
  mfs_export("solid.pos", "msh", wings, "statresp",
             "disp", "stress", "resultant");

  mfs_merge("solid.pos", "aero.pos", "wing.dsp", "msh");
```

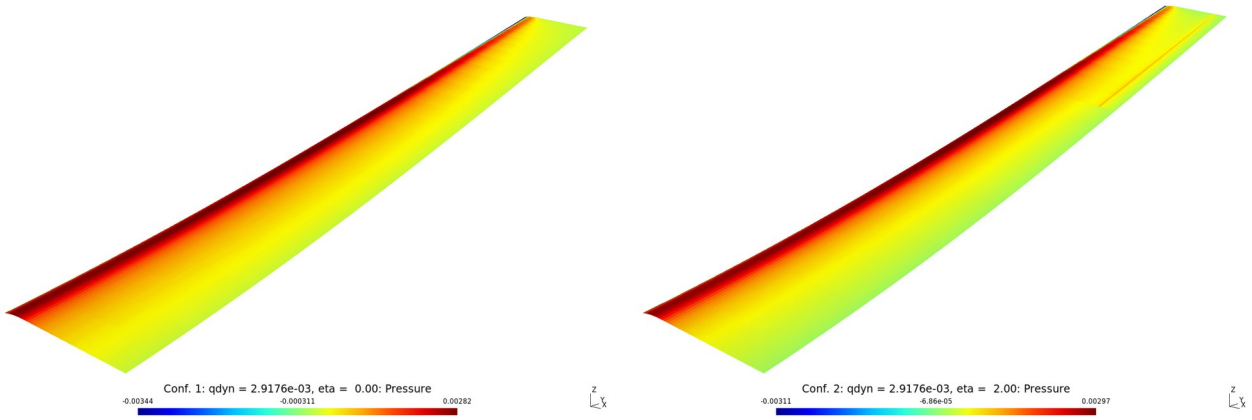
Figure 2.2-10 shows the deformation of the wing and the aerodynamic pressure for the two configurations.

Next, we retrieve the pressure in the four wing sections already considered in the analysis of the rigid wing, compute the pressure coefficient and compare the results of the flexible wing with those of the rigid wing.

```
# Compare pressure in wing sections with rigid results

  load rigres.bin
  pri = rigres.pi; pro  = rigres.po;

  [xi(:, 1), pfi(:, :, 1), y(1)] = ...
     mfs_xydata(winga, "statresp", "pressure", 1, ycoli(1));
  [xi(:, 2), pfi(:, :, 2), y(2)] = ...
     mfs_xydata(winga, "statresp", "pressure", 1, ycoli(2));
  [xo(:, 1), pfo(:, :, 1), y(3)] = ...
     mfs_xydata(winga, "statresp", "pressure", [2, 3], ycolo(1));
  [xo(:, 2), pfo(:, :, 2), y(4)] = ...
```



Configuration 1 Configuration 2

Figure 2.2-10: Pressure on Deformed Wing (MPa)

```
    mfs_xydata(winga, "statresp", "pressure", [2, 3], ycolo(2));

pfi = pfi / qdyn; pfo = pfo / qdyn;
xi  = xi / cr;    xo  = xo / cr;
y   = y * 1e-3;

for m = 1 : 2

   for n = 1 : length(y)
       header{n} = sprintf("\\eta = %1d\\deg, y = %4.2f m",
                           eta(m), y(n));
   endfor

   figure(m, "position", [100 * m, 500, 750, 500],
             "paperposition", [0, 0, 17, 12]);
   subplot(2, 2, 1)
      plot(xi(:, 1), squeeze(pri(:, m, 1)),
           xi(:, 1), squeeze(pfi(:, m, 1)));
      legend("rigid", "flexible", "location", "southwest");
      title(header{1});
      grid;
      ylim([-1, 1]);
      ylabel('\Delta c_P');
   subplot(2, 2, 2)
      plot(xi(:, 2), squeeze(pri(:, m, 2)),
```

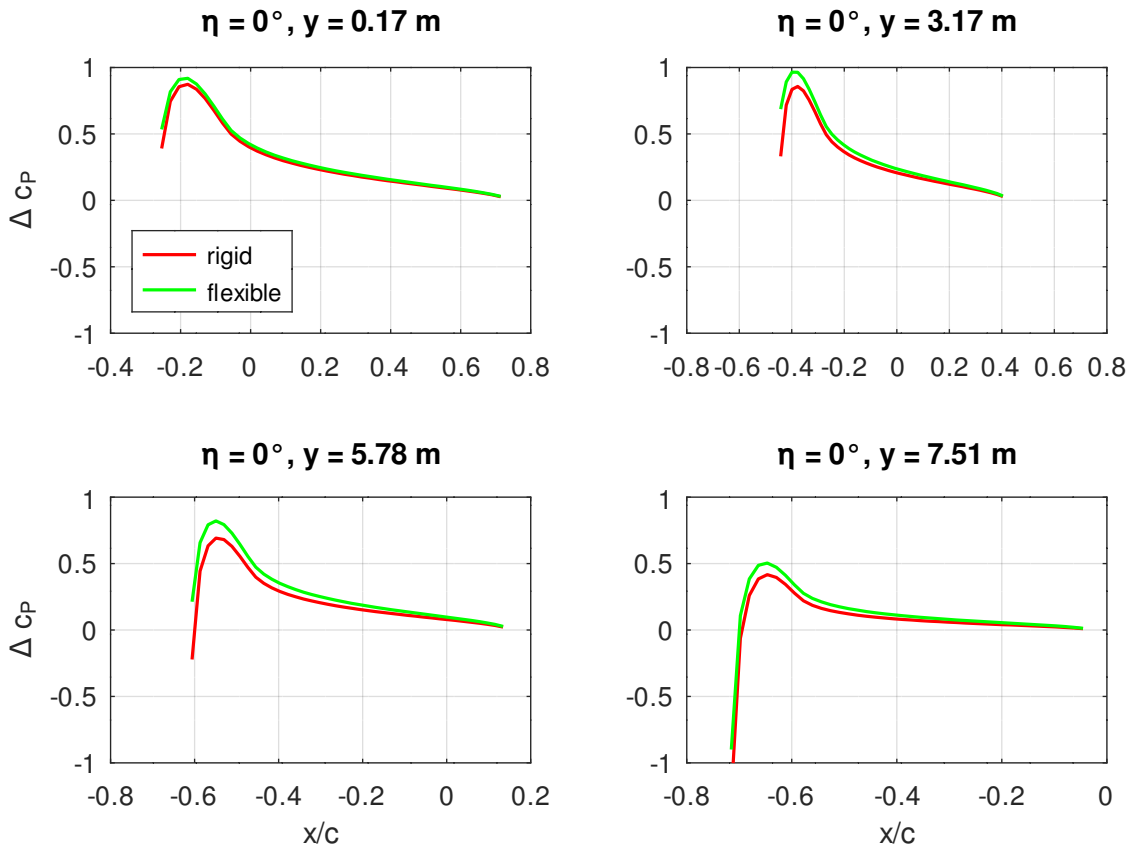


Figure 2.2-11: Comparison of Pressure Coefficients, Configuration 1

```
            xi(:, 2), squeeze(pfi(:, m, 2)));
      title(header{2});
      grid;
      ylim([-1, 1]);
   subplot(2, 2, 3)
      plot(xo(:, 1), squeeze(pro(:, m, 1)),
           xo(:, 1), squeeze(pfo(:, m, 1)));
      title(header{3});
      grid;
      ylim([-1, 1]);
      xlabel('x/c'); ylabel('\Delta c_P');
   subplot(2, 2, 4)
      plot(xo(:, 2), squeeze(pro(:, m, 2)),
           xo(:, 2), squeeze(pfo(:, m, 2)));
      title(header{4});
      grid;
      ylim([-1, 1]);
      xlabel('x/c');
   print(["cp", num2str(m), EXT], FORMAT);

endfor
```

Figure 2.2-11 compares the pressure coefficients of configuration 1 and Figure 2.2-12 those of configuration 2. In both configurations, the pressure coef-

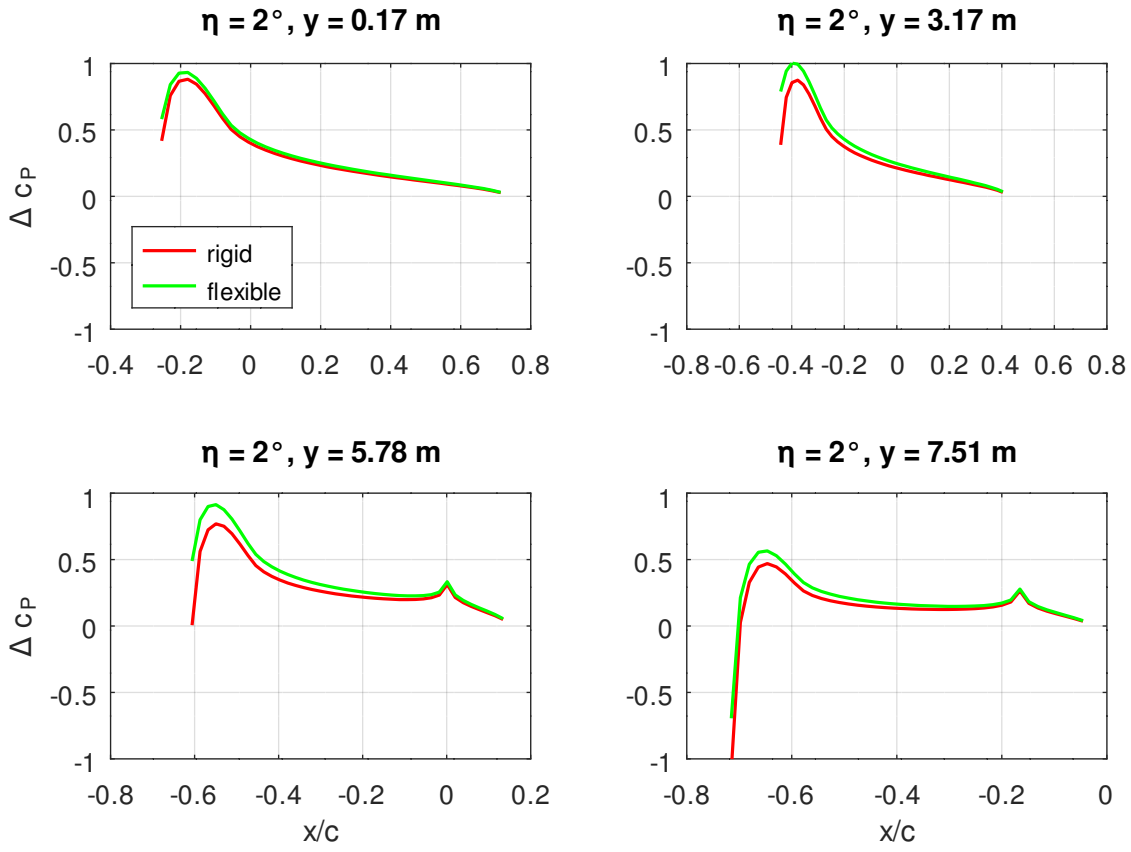


Figure 2.2-12: Comparison of Pressure Coefficients, Configuration 2

ficient of the flexible wing is slightly larger than that of the rigid wing. This confirms that the effect of bending, which increases the effective angle of attack, is greater than the effect of torsion, which reduces the angle of attack.

Next, we compute the lift coefficient in spanwise direction. The lift coefficient is obtained by dividing the lift per length by the reference length and the dynamic pressure.

```
# Compare lift coefficient in spanwise direction

  [s, f, m] = mfs_xydata(winga, "statresp", "spanload",
                         {1, [2, 3]});
  clflex = squeeze(f(3, :, :)) / (cr * qdyn);
  clrig  = rigres.cl;
  s     *= 1e-3;

  figure(3, "position", [300, 500, 750, 500],
            "paperposition", [0, 0, 15, 10]);
  plot(s, clflex,
       s, clrig, "linestyle", "--");
  legend("Conf. 1 flex.", "Conf. 2 flex.",
         "Conf. 1 rig.",  "Conf. 2 rig.",
         "location", "southwest");
  grid
  xlabel('s [m]'); ylabel('c_l');
  print(["cl", EXT], FORMAT);
```

Figure 2.2-13 compares the spanwise lift coefficient of the flexible wing with

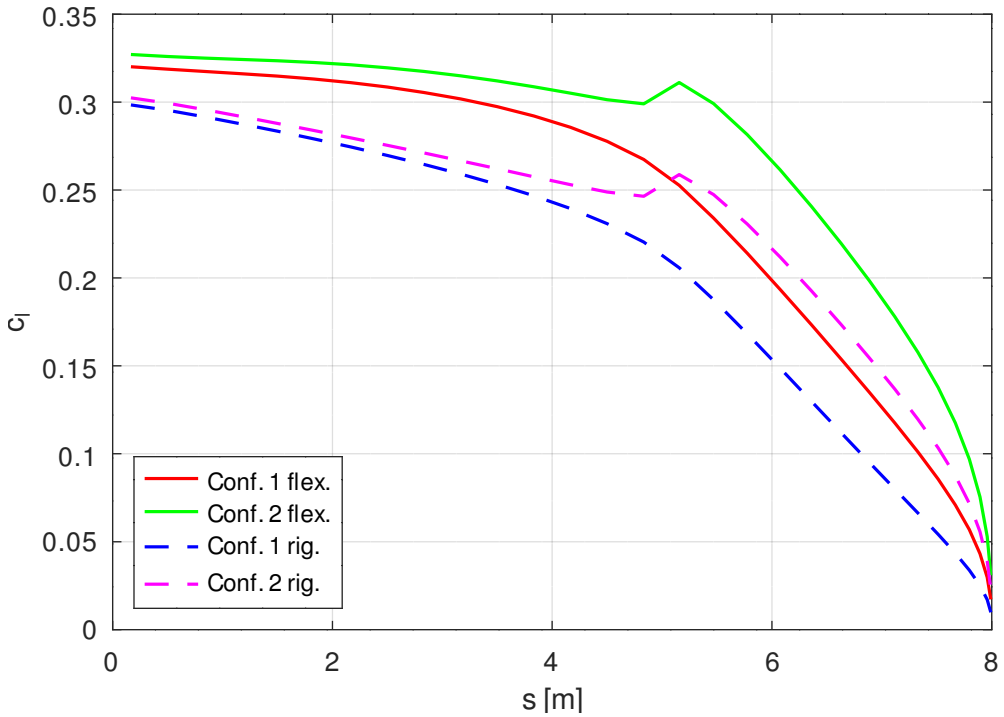


Figure 2.2-13: Comparison of Spanwise Lift Coefficients

that of the rigid wing. It can be seen that the lift coefficient of the flexible wing is larger than that of the rigid wing. This observation agrees with the results shown in Figure 2.2-9.

Next, we look at the stress resultants in the stringers. The stringers are essentially loaded by normal forces and bending moments. We expect the largest loads to occur in the stringers of the two spars, i.e. in the upper and lower stringers of the second and third row, cf. Figure 2.2-3.

Function **mfs_getresp** returns a cell array of structures, where each structure contains the results of one element. Because all elements are of the same type, all structures have the same fields. The cell arrays can therefore be converted into structure arrays. The script extracts the *y*-coordinate from field **coor** and subsequently plots the normal force N and the bending moment M_y about the element y_E-axis as a function of the global *y*-coordinate. In addition, the stress resultants in the lower stringers of the second row are written to the output file.

```
# Stress resultants in stringers

  RU2 = mfs_getresp(wings, "statresp", "resultant",
                    "Stringer_2_Upper", 2);
  RU3 = mfs_getresp(wings, "statresp", "resultant",
                    "Stringer_3_Upper", 2);
  RL2 = mfs_getresp(wings, "statresp", "resultant",
                    "Stringer_2_Lower", 2);
  RL3 = mfs_getresp(wings, "statresp", "resultant",
                    "Stringer_3_Lower", 2);
  nelt = length(RU2);

  RU2 = cell2mat(RU2); RU3 = cell2mat(RU3);
  RL2 = cell2mat(RL2); RL3 = cell2mat(RL3);

  coor = reshape([RU2.coor], 3, nelt);
  y    = coor(2, :) * 1e-3;

  figure(3, "position", [300, 500, 750, 500],
            "paperposition", [0, 0, 17, 12]);
  subplot(2, 1, 1)
     plot(y, [RU2.N] * 1e-3, y, [RU3.N] * 1e-3,
          y, [RL2.N] * 1e-3, y, [RL3.N] * 1e-3);
     legend("2 Upper", "3 Upper", "2 Lower", "3 Lower");
     legend("numcolumns", 2);
     grid;
     ylabel('kN [N]');
  subplot(2, 1, 2);
     plot(y, [RU2.My] * 1e-3, y, [RU3.My] * 1e-3,
          y, [RL2.My] * 1e-3, y, [RL3.My] * 1e-3);
     grid;
     xlabel('y [m]'); ylabel('M_y [Nm]');
  print(["resultants", EXT], FORMAT);
```

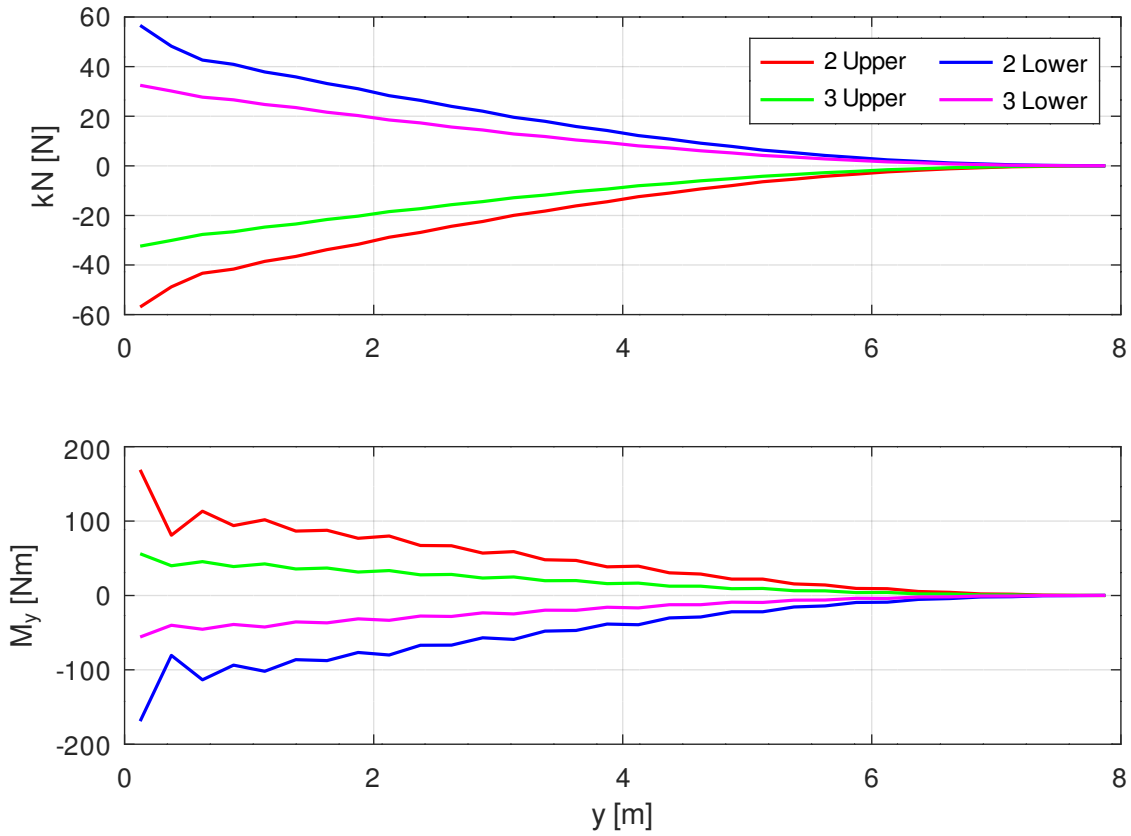


Figure 2.2-14: Normal Forces and Bending Moments in Stringers

```
mfs_print(fid, wings, "statresp",
          {"resultant", "Stringer_2_Lower"});
```

Figure 2.2-14 shows the resulting diagrams. As expected the normal forces are positive in the lower stringers and negative in the upper stringers. The bending moments refer to the local y_E-axis of the element coordinate system. The local y_E-axis of the upper stringers points backwards and that of the lower stringers points forwards. Therefore, the bending moments in the upper and lower stringers have a different sign.

The diagram shows that the stress resultants in the stringers of the second row, i.e. the stringers of the main spar, are larger than those in the stringers of the third row. Therefore, only the stress resultants in the lower stringers of the main spar are written to the output file.

```
Component "wings"

Stress resultants of loadcase  1

  Beam elements:

  element         N            Qy            Qz            Mx            My            Mz
  ------------------------------------------------------------------------------------------
      145     4.696e+04    -1.112e+02     1.149e+02    -2.156e+02    -1.412e+05    -6.527e+03
      146     3.978e+04     8.249e+01    -1.335e+02     6.563e+01    -6.555e+04     6.728e+03
```

147	3.486e+04	−5.292e+01	2.715e+01	5.362e+01	−9.329e+04	−5.133e+03
148	3.330e+04	3.971e+01	−2.154e+01	8.588e+01	−7.561e+04	−7.496e+02
153	3.055e+04	−5.782e+00	1.562e+01	6.175e+01	−8.278e+04	−1.500e+03
154	2.880e+04	3.907e+00	−2.000e+01	6.281e+01	−6.901e+04	−1.219e+03
155	2.642e+04	−1.043e+01	−7.158e+00	8.429e+01	−7.002e+04	−2.366e−01
156	2.458e+04	1.779e+01	−3.899e+00	8.865e+01	−6.023e+04	−3.161e+03
161	2.215e+04	4.924e+00	9.351e+00	5.209e+01	−6.315e+04	1.010e+02
162	2.053e+04	−1.387e+00	−1.569e+01	5.782e+01	−5.188e+04	−1.502e+03
163	1.844e+04	−6.064e+00	−7.494e+00	6.904e+01	−5.164e+04	2.030e+02
164	1.682e+04	1.232e+01	−2.545e+00	7.372e+01	−4.308e+04	−2.417e+03
169	1.477e+04	4.204e+00	6.500e+00	4.360e+01	−4.484e+04	1.527e+02
170	1.339e+04	−1.472e+00	−1.249e+01	4.820e+01	−3.556e+04	−1.086e+03
171	1.169e+04	−3.966e+00	−6.508e+00	5.308e+01	−3.479e+04	9.138e+01
172	1.038e+04	7.908e+00	−1.702e+00	5.571e+01	−2.771e+04	−1.580e+03
177	8.762e+03	2.744e+00	4.136e+00	3.329e+01	−2.852e+04	8.960e+01
178	7.672e+03	−1.128e+00	−9.243e+00	3.591e+01	−2.133e+04	−6.727e+02
179	6.396e+03	−2.278e+00	−4.999e+00	3.621e+01	−2.027e+04	2.469e+00
180	5.430e+03	4.287e+00	−1.109e+00	3.648e+01	−1.500e+04	−8.822e+02
	…	…	…	…	…	…

The output shows the the predominant stress resultants are the normal force and the bending moments.

Figure 2.2-14 shows that the largest values of the stress resultants occur at the wing root. Thus it is sufficient to look at the stresses in the beam elements at the wing root.

Figure 2.2-15 shows that the element numbers increase along the stringers. The numbers of the elements at the wing root are 141 and 145 for the stringers of the main spar and 205 and 209 for the stringers of the back spar. These element numbers can also be accessed from the element sets.

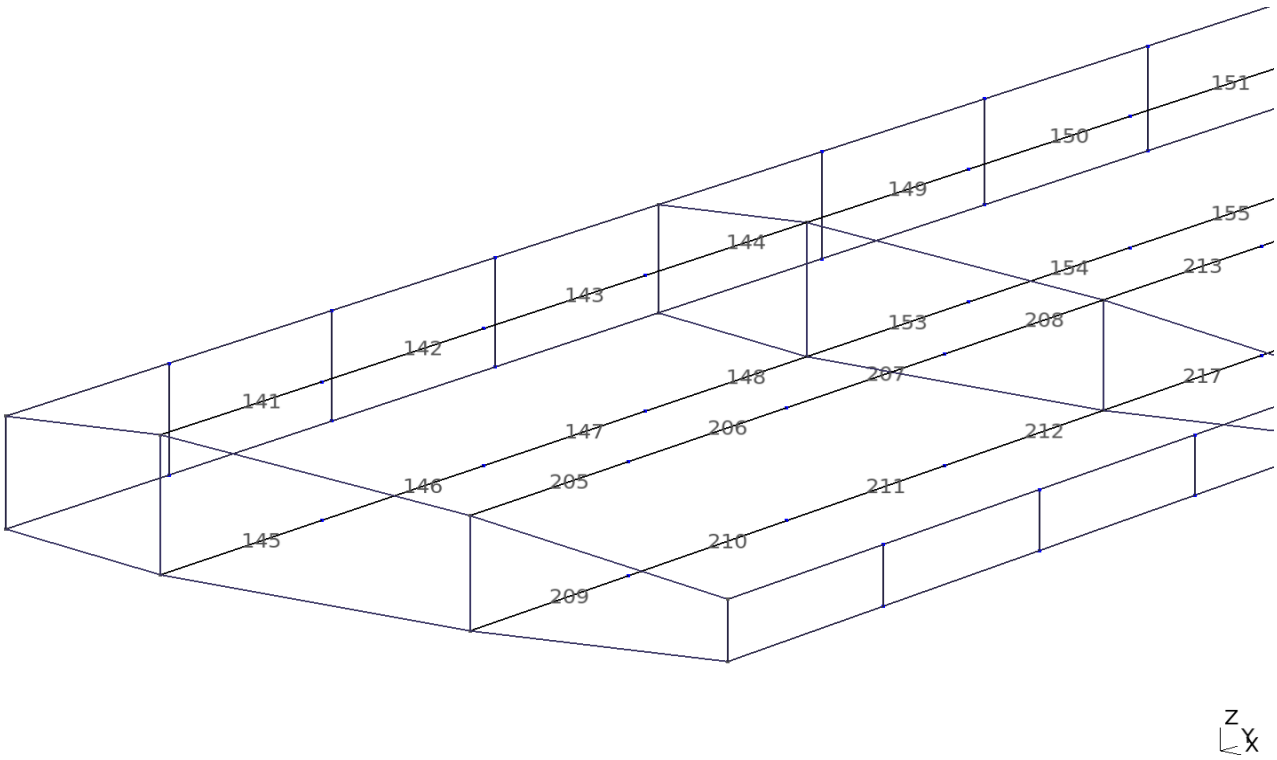


Figure 2.2-15: Element Numbers of Stringers

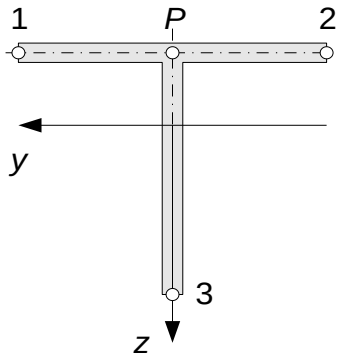


Figure 2.2-16: Stress Points

The cross sections of the beams can be considered thin walled. Thus, it is sufficient to compute the stresses at the three points shown in Figure 2.2-16. Mefisto function **mfs_beamstress** can be used to compute normal stresses in beams due to normal forces and bending moments. It is sufficient to consider only configuration 2.

```
# Stresses in beams

  load data.bin

  for n = 2 : 3
      Upper = sprintf("Stringer_%d_Upper", n);
      Lower = sprintf("Stringer_%d_Lower", n);
      elemU = mfs_getset(wings, "eset", Upper);
      elemL = mfs_getset(wings, "eset", Lower);
      y1 = 0.5 * stringer(n).b;
      zo = data.(Upper).geom.P(2);
      zu = zo + stringer(n).h - 0.5 * stringer(n).t;
      coor = [y1, zo; -y1, zo; 0, zu];
      mfs_beamstress(fid, wings, "statresp", [elemU(1), elemL(1)],
                     coor, 2);
  endfor

  fclose(fid);
```

The stresses are computed at the midpoint of the beam and written to the output file.

```
--------------------------------------------------------------------------------

Component "wings"

Beam stresses of loadcase 2

  element  point      y           z          sig
  ------------------------------------------------
      141      1   2.500e+01  -1.037e+01  -2.000e+02
               2  -2.500e+01  -1.037e+01  -2.020e+02
               3   0.000e+00   2.763e+01  -7.096e+01
      145      1   2.500e+01  -1.037e+01   2.050e+02
               2  -2.500e+01  -1.037e+01   1.952e+02
               3   0.000e+00   2.763e+01   6.992e+01
```

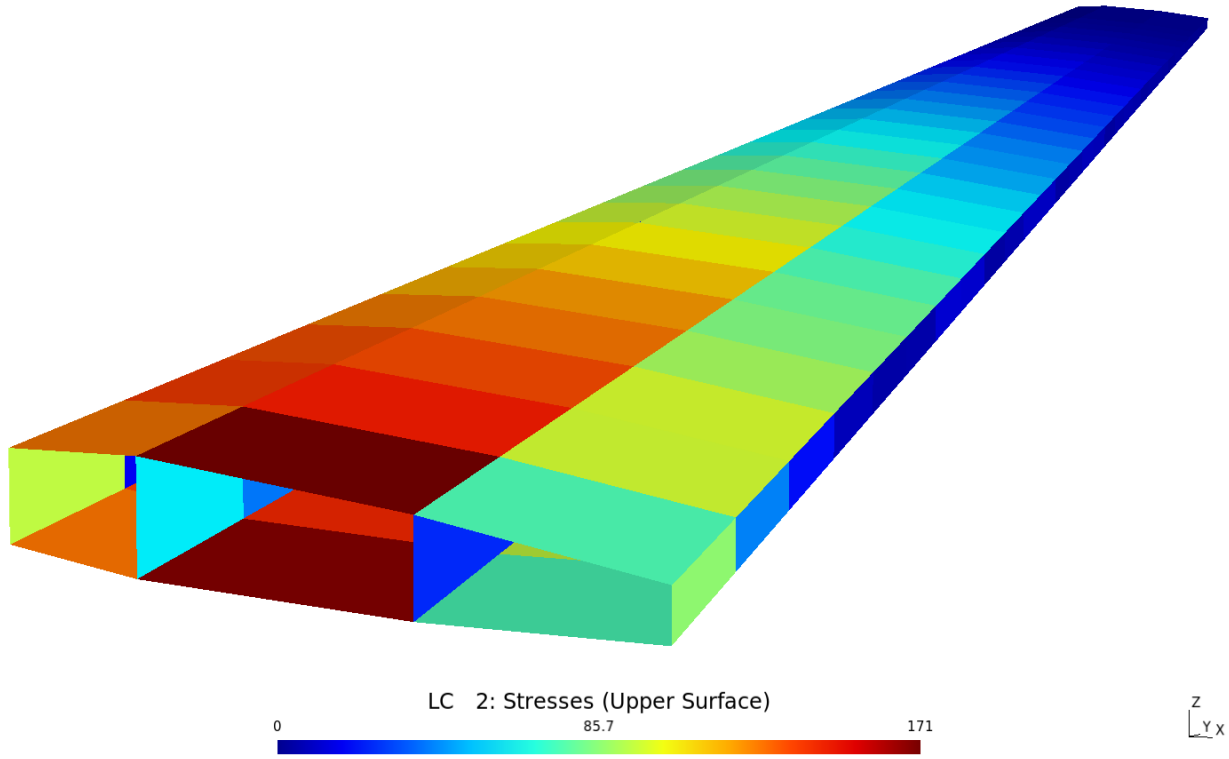


Figure 2.2-17: Von Mises Stress in Shell Elements (MPa)

```
-------------------------------------------------------------------------------

Component "wings"

Beam stresses of loadcase 2

  element  point      y           z          sig
  ---------------------------------------------------
      205      1   2.000e+01  -7.909e+00  -1.485e+02
               2  -2.000e+01  -7.909e+00  -1.405e+02
               3   0.000e+00   2.009e+01  -6.720e+01
      209      1   2.000e+01  -7.909e+00   1.433e+02
               2  -2.000e+01  -7.909e+00   1.463e+02
               3   0.000e+00   2.009e+01   6.753e+01
```

The largest normal stress is $205\ \mathrm{MPa}$. It occurs at the wing root in the lower stringer of the main spar. The slightly different stress values at points 1 and 2 indicate that there is also a small bending moment about the element z_E-axis.

Finally we look at the stresses in the shell elements. Figure 2.2-17 shows the von Mises stresses in the upper surface of the shell elements for configuration 2. As expected, the maximal stress occurs at the wing root. The value of the maximum is $171\ \mathrm{MPa}$.

3 Trim Analysis

3.1 Glider

Summary

Directory:	`exa/aeroelastic/trim/glider`
Objectives:	• learn how to perform a trim analysis of a flexible aircraft • understand the differences between restrained and unrestrained trim analysis • learn how to access trim parameters • learn how to access stress resultants
Elements:	**`b2, m1`**
Method:	Vortex-Lattice
Functions:	**`mfs_line, mfs_linenodes, mfs_beamsection, mfs_airfoil, mfs_new, mfs_stiff, mfs_mass, mfs_massproperties, mfs_freevib, mfs_splines, mfs_trim, mfs_results, mfs_export, mfs_print, mfs_getresp, mfs_xydata, mfs_merge`**

Problem Description

Perform a trim analysis of the standard class glider shown in Figure 3.1-1. Consider the following manoeuvres:

1. Straight flight with a velocity of $30\ \mathrm{m/s}$
2. Truly banked left turn with a bank angle of 30° and a velocity of $30\ \mathrm{m/s}$
3. Left turn with a bank angle of 30° and zero side slip
4. Sudden aileron deflection to 1° at a velocity of $30\ \mathrm{m/s}$

The wing has a FX 66-S-196 V1 airfoil the data of which can be found at Airfoil Tools. The rigging angle of incidence linearly decreases from 0° at the wing root to -2° at the wing tip. The dihedral angle is 2°.

The vertical and the horizontal stabilizer have symmetric airfoils. The rigging angle of incidence of the horizontal stabilizer is -3°.

The flap ratio of the aileron is 20 %. The depth of the rudder is $200\ \mathrm{mm}$ and that of the elevator $100\ \mathrm{mm}$.

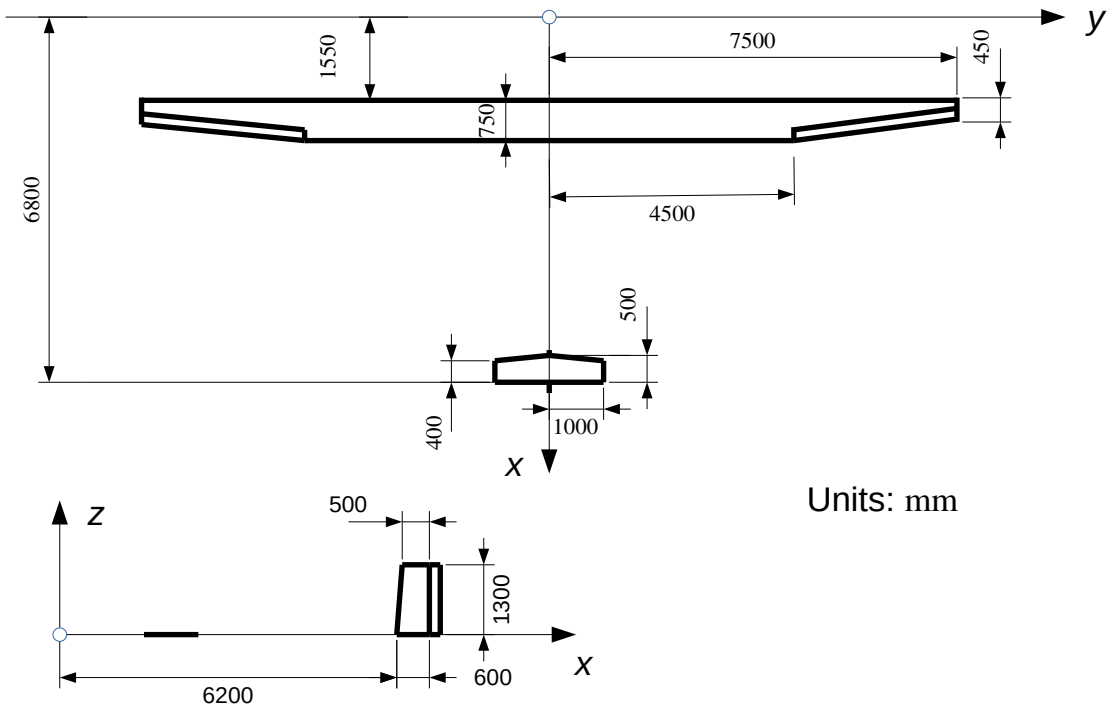


Figure 3.1-1: Geometry of Standard Class Glider

The mass density of the air is $1.21\ \mathrm{kg/m^3}$.

The solid structure is represented by a beam model, with a point mass for the pilot and an additional point mass at the tip of the glider which is used as a trim mass. Figure 3.1-2 shows the solid model of the glider.

The cross sections of the beams are as follows:

Fuselage:	Cabin	Thin rings with varying radii
	Tail	Thin rings with varying radii
Wing:	Spar	Thin box
	Leading edge	Bar
	Trailing edge	Bar
Vertical Stabilizer:	Front	Bar
	Rear	Thin box
	Horizontal	Bar
Horizontal Stabilizer:	Front	Thin box
	Rear	Thin box

Tips Bar

The cross section data can be found in file `solid.m`.

Parameters

Parameters used by more than one script are defined in file `params.m`. The units used are kg, m and s.

```
# ====================================================================
# Data needed by several scripts (kg, m, s)
# ====================================================================

# Geometry

  b     =   15;  % Span of wing
  cr    = 0.75;  % Chord length of wing at wing root
  ct    = 0.45;  % Chord length of wing at wing tip
  delta =    2;  % Dihedral angle in degrees
  lr    =  0.4;  % Aileron length ratio
  fr    =  0.2;  % Aileron flap ratio
  at    =   -2;  % Angle of incidence at wing tip (in degrees)
  l     =  6.8;  % Length of fuselage
  be    =  2.0;  % Span of elevator
  h     =  1.3;  % Height of fin
```

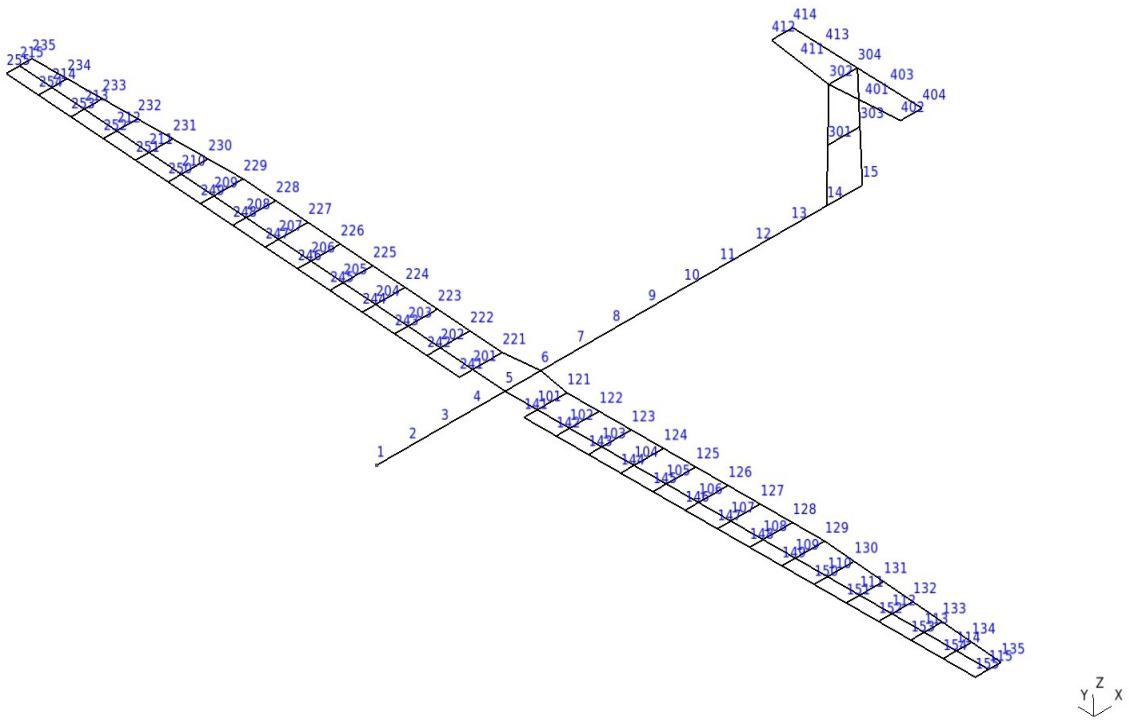


Figure 3.1-2: Solid Model of the Glider

```
  xle   = 1.55;  % x-position of leading edge
  ds    =  1.0;  % Distance of yaw string from center of mass

# Environment and Manoeuvres

  v     =   30;   % Flight velocity
  rho   = 1.21;   % Mass density of air
  g     = 9.81;   % Gravity acceleration
  gamma =   30;   % Bank angle in degrees
  eta   =    1;   % Aileron deflection in degrees
```

3.1.1 Solid Model

GNU Octave script `solid.m` defines the solid model. The solid component is saved to file `solid.bin`. In addition, the mass properties are saved to file `solid_mass.bin`.

Model Definition

The script first loads the parameters from file `params.m` and defines the material data and the number of modes to compute.

```
# Example: Trim analysis of a standard class glider
#
# a) Solid Model
#
# ------------------------------------------------------------

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

# ============================================================
# Data (kg, m, s)
# ============================================================

# Geometry

  params

# Point masses

  m   =  85;   % Mass of the pilot
  mt  =   5;   % Trim mass

# Material

  mat = struct("type", "iso", "E",   4E10,
               "ny",     0.3, "rho", 2000);

# Analysis

  nofmod = 5;  % Number of normal modes
```

Next, we compute some coordinates:

```
# x-positions

  xwr = xle + 0.25 * cr;          % Wing root
  xtr = xle + (1 - fr) * cr;      % Trailing edge of inner wing
  xtt = xle + (1 - fr) * ct;      % Trailing edge at wing tip

# y-positions

  yt = 0.5 * b;                   % Wing tip
  ya = yt * (1 - lr);             % Begin of aileron
  y1 = yt / 15;                   % First rib

# z-positions

  dihedral = tand(delta);
  zt = yt * dihedral;             % Wing tip
  za = ya * dihedral;             % Begin of aileron
  z1 = y1 * dihedral;             % First rib
```

The model definition begins with the definition of the model type and the subtype:

```
# ================================================================
# Model Definition
# ================================================================

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

Then, we define the nodes and elements of the cabin and the tail. Node identifiers range from 1 to 15 and element identifiers from 1 to 14. Function **mfs_linenodes** is used to define the nodes.

```
# Fuselage
# --------

# Cross section properties: Radius and thickness

  fuselage.r = [0.2000, 0.3000, 0.4000, 0.4000, 0.3667, ...
                0.2667, 0.1367, 0.1140, 0.1100, 0.1080, ...
                0.1060, 0.1040, 0.1020, 0.1000];
  fuselage.t = [ 0.005,  0.006,  0.006,  0.005, 0.005, ...
                 0.004,  0.003,  0.003,  0.003, 0.003, ...
                 0.003,  0.003,  0.003,  0.003];

# Cabin nodes

  nodes(1).id = 1; nodes(1).coor = [  0, 0, 0]; % Tip
  nodes(2).id = 5; nodes(2).coor = [xwr, 0, 0]; % Wing root

  nodes = mfs_linenodes(nodes, 1, 5, 2 : 4);
```

```
# Tail nodes

  nodes(6).id = 15; nodes(6).coor = [l, 0, 0];  % End Node
  nodes = mfs_linenodes(nodes, 5, 15, 6 : 14);

# Elements

  for k = 1 : 14
      geom = mfs_beamsection("ring", "thin",
                             fuselage.r(k), fuselage.t(k));
      geom.v = [0, 0, 1];
      elem(k) = struct("id", k, "type", "b2",
                       "nodes", [k, k + 1],
                       "geom", geom, "mat", mat);
  endfor
```

Subsequently, we define the nodes and elements of the wing. We begin with the definition of the cross section data.

```
# Wing beam sections
# ------------------

  geom_spar   = mfs_beamsection("box", "thin", 0.4, 0.2, 0.003);
  geom_spar.v = [0, 0, 1];

  geom_te   = mfs_beamsection("bar", 0.001, 0.05);
  geom_te.v = [0, 0, 1];

  geom_le   = mfs_beamsection("bar", 0.001, 0.05);
  geom_le.v = [0, 0, 1];

  geom_ribs   = mfs_beamsection("I", 0.1, 0.1, 0.002, 0.002);
  geom_ribs.v = [0, 0, 1];
```

Then, we define the nodes and elements of the left wing. Node and element identifiers range from 101 to 155.

```
# Left Wing
# ---------

# Spar

  nnode = length(nodes) + 1;
  nodes(nnode).id = 109; nodes(nnode++).coor = [xwr, -ya, za];
  nodes(nnode).id = 115; nodes(nnode).coor = [xwr, -yt, zt];
  [nodes, ells1] = mfs_line(nodes, 5, 109, 101 : 108, 101 : 109,
                            "b2", geom_spar, mat);
  [nodes, ells2] = mfs_line(nodes, 109, 115, 110 : 114, 110 : 115,
                            "b2", geom_spar, mat);
  eset.left_spar = 101 : 115;

# Trailing Edge
```

```
  nnode = length(nodes) + 1;
  nodes(nnode).id = 121; nodes(nnode++).coor = [xtr, -y1, z1];
  nel = length(elem) + 1;
  elem(nel).id = 121; elem(nel).type = "b2";
  elem(nel).nodes = [6, 121];
  elem(nel).geom = geom_te; elem(nel).mat = mat;

  nodes(nnode).id = 129; nodes(nnode++).coor = [xtr, -ya , za];
  nodes(nnode).id = 135; nodes(nnode).coor = [xtt, -yt, zt];
  [nodes, ellt1] = mfs_line(nodes, 121, 129, 122 : 128, 122 : 129,
                            "b2", geom_te, mat);
  [nodes, ellt2] = mfs_line(nodes, 129, 135, 130 : 134, 130 : 135,
                            "b2", geom_te, mat);

# Leading edge

  nnode = length(nodes) + 1;
  nodes(nnode).id = 141; nodes(nnode++).coor = [xle, -y1, z1];
  nodes(nnode).id = 149; nodes(nnode++).coor = [xle, -ya, za];
  nodes(nnode).id = 155; nodes(nnode).coor = [xle, -yt, zt];
  [nodes, elll1] = mfs_line(nodes, 141, 149, 142 : 148, 142 : 149,
                            "b2", geom_le, mat);
  [nodes, elll2] = mfs_line(nodes, 149, 155, 150 : 154, 150 : 155,
                            "b2", geom_le, mat);

  elem = [elem, ells1, ells2, ellt1, ellt2, elll1, elll2];

# Ribs

  nel1 = length(elem) + 1;
  nel2 = nel1 + 14;

  n1 = 141; n2 = 101; n3 = 121; ne = nel1; ide = 161;
  for k = nel1 : nel2
      elem(ne++) = struct("id", ide++, "type", "b2",
                          "nodes", [n1++, n2],
                          "geom", geom_ribs, "mat", mat);
      elem(ne++) = struct("id", ide++, "type", "b2",
                          "nodes", [n2++, n3++],
                          "geom", geom_ribs, "mat", mat);
  endfor

  nset.left_wing = [5 : 6, 101 : 115, 121 : 135, 141 : 155];
```

Similarly, the nodes and elements of the right wing are defined. Node and element identifiers range from 201 to 255.

```
# Right wing
# ----------

# Spar

  nnode = length(nodes) + 1;
  nodes(nnode).id = 209; nodes(nnode++).coor = [xwr, ya, za];
```

```
  nodes(nnode).id = 215; nodes(nnode).coor = [xwr, yt, zt];
  [nodes, elrs1] = mfs_line(nodes, 5, 209, 201 : 208, 201 : 209,
                            "b2", geom_spar, mat);
  [nodes, elrs2] = mfs_line(nodes, 209, 215, 210 : 214, 210 : 215,
                            "b2", geom_spar, mat);
  eset.right_spar = 201 : 215;

# Trailing Edge

  nnode = length(nodes) + 1;
  nodes(nnode).id = 221; nodes(nnode++).coor = [xtr,  y1, z1];
  nel = length(elem) + 1;
  elem(nel).id = 221; elem(nel).type = "b2";
  elem(nel).nodes = [6, 221];
  elem(nel).geom = geom_te; elem(nel).mat = mat;

  nodes(nnode).id = 229; nodes(nnode++).coor = [xtr, ya, za];
  nodes(nnode).id = 235; nodes(nnode).coor = [xtt, yt, zt];
  [nodes, elrt1] = mfs_line(nodes, 221, 229, 222 : 228, 222 : 229,
                            "b2", geom_te, mat);
  [nodes, elrt2] = mfs_line(nodes, 229, 235, 230 : 234, 230 : 235,
                            "b2", geom_te, mat);

# Leading edge

  nnode = length(nodes) + 1;
  nodes(nnode).id = 241; nodes(nnode++).coor = [xle,  y1, z1];
  nodes(nnode).id = 249; nodes(nnode++).coor = [xle,  ya, za];
  nodes(nnode).id = 255; nodes(nnode).coor = [xle,  yt, zt];
  [nodes, elrl1] = mfs_line(nodes, 241, 249, 242 : 248, 242 : 249,
                            "b2", geom_le, mat);
  [nodes, elrl2] = mfs_line(nodes, 249, 255, 250 : 254, 250 : 255,
                            "b2", geom_le, mat);

  elem = [elem, elrs1, elrs2, elrt1, elrt2, elrl1, elrl2];

# Ribs

  nel1 = length(elem) + 1;
  nel2 = nel1 + 14;

  n1 = 241; n2 = 201; n3 = 221; ne = nel1; ide = 261;
  for k = nel1 : nel2
      elem(ne++) = struct("id", ide++, "type", "b2",
                          "nodes", [n1++, n2],
                          "geom", geom_ribs, "mat", mat);
      elem(ne++) = struct("id", ide++, "type", "b2",
                          "nodes", [n2++, n3++],
                          "geom", geom_ribs, "mat", mat);
  endfor

  nset.right_wing = [5 : 6, 201 : 215, 221 : 235, 241 : 255];
```

The vertical stabilizer consists of a front beam, a rear beam and a horizontal

beam joining both. Node identifiers range from 301 to 304 and element identifiers from 301 to 305.

```
# Vertical stabilizer
# -------------------

  geom  = mfs_beamsection("box", "thin", 0.10, 0.4, 0.002);
  geom.v = [1, 0, 0];

  nnode = length(nodes) + 1;
  nodes(nnode).id = 304; nodes(nnode).coor = [l, 0, h];
  [nodes, elvsr] = mfs_line(nodes, 15, 304, 303, 303 : 304,
                            "b2", geom, mat);

  geom  = mfs_beamsection("bar", 0.05, 0.002);
  geom.v = [1, 0, 0];

  nnode = length(nodes) + 1;
  nodes(nnode).id = 302; nodes(nnode).coor = [l - 0.4, 0, h];
  [nodes, elvsf] = mfs_line(nodes, 14, 302, 301, 301 : 302,
                            "b2", geom, mat);

  elem = [elem, elvsr, elvsf];

  geom  = mfs_beamsection("bar", 0.05, 0.002);
  geom.v = [0, 0, 1];

  nel = length(elem) + 1;
  id1 = 301; id2 = 303; ide = 305;
  for k = 1 : 2
      elem(nel++) = struct("id", ide++, "type", "b2",
                           "nodes", [id1++, id2++],
                           "geom", geom, "mat", mat);
  endfor

  nset.vertical = [14, 15, 301 : 304];
```

Finally, the horizontal stabilizer consists of two front and rear beams and two beams at both ends. Node identifiers range from 401 to 404 for the left part and from 411 to 414 for the right part. Element identifiers range from 401 to 422.

```
# Horizontal stabilizer (root is at node 302)
# -------------------------------------------

  d = 0.5 * be;

  geomr   = mfs_beamsection("box", "thin", 0.3, 0.08, 0.002);
  geomr.v = [0, 0, 1];

  geomf   = mfs_beamsection("box", "thin", 0.1, 0.05, 0.002);
  geomf.v = [0, 0, 1];

  nnode = length(nodes) + 1;
```

```
  nodes(nnode).id = 402; nodes(nnode++).coor = [l - 0.3, -d, h];
  nodes(nnode).id = 412; nodes(nnode++).coor = [l - 0.3,  d, h];
  nodes(nnode).id = 404; nodes(nnode++).coor = [l, -d, h];
  nodes(nnode).id = 414; nodes(nnode++).coor = [l,  d, h];

  [nodes, elhsfl] = mfs_line(nodes, 302, 402, 401, 401 : 402,
                             "b2", geomf, mat);
  [nodes, elhsrl] = mfs_line(nodes, 304, 404, 403, 403 : 404,
                             "b2", geomr, mat);

  [nodes, elhsfr] = mfs_line(nodes, 302, 412, 411, 411 : 412,
                             "b2", geomf, mat);
  [nodes, elhsrr] = mfs_line(nodes, 304, 414, 413, 413 : 414,
                            "b2", geomr, mat);

  elem = [elem, elhsfl, elhsrl, elhsfr, elhsrr];

  geom   = mfs_beamsection("bar", 0.002, 0.04);
  geom.v = [0, 0, 1];

  nel = length(elem) + 1;
  elem(nel++) = struct("id", 421, "type", "b2",
                       "nodes", [402, 404],
                       "geom", geom, "mat", mat);
  elem(nel++) = struct("id", 422, "type", "b2",
                       "nodes", [412, 414],
                       "geom", geom, "mat", mat);

  nset.horizontal = [302, 304, 401 : 404, 411 : 414];
```

A point mass is used to add the mass of the pilot. It is connected to node 3. In addition, a further small point mass used as trim mass is connected to node 1.

```
# Pilot
# -----

  mass.m = m;
  elem(nel++) = struct("id",  500, "type", "m1",
                       "nodes", 3, "geom", mass, "mat", []);

# Trim mass
# ---------

  mass.m = mt;
  elem(nel) = struct("id",  510, "type", "m1",
                     "nodes", 1, "geom", mass, "mat", []);

  model.nodes    = nodes; model.elements = elem;
  model.nset     = nset;  model.eset     = eset;
```

To prevent rigid body motions, six translations are constrained at nodes 6, 101 and 201:

```
# Constraints
# -----------

  prescribed = struct("id",    {  6,    101, 201},
                      "dofs", {1 : 3, [1, 3], 3});

  model.constraints.prescribed = prescribed;
```

Analysis

First, the solid component is created. Subsequently, the stiffness and mass matrices are computed. The mass properties are written to the output file.

```
# ============================================================
# Analysis
# ============================================================

# Create and export component

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

# Matrices

  gliders = mfs_stiff(gliders);
  gliders = mfs_mass(gliders);
  mp = mfs_massproperties(fid, gliders);

  save -binary solid.bin gliders
  save -binary solid_mass.bin mp

  fclose(fid);
```

Results

The output file contains the following results:

```
Mefisto 2.8: Building new component from input "model"

  Model Type = solid, Model Subtype = 3d

  Number of nodes      =   117,  Number of elements =   180
  Number of element types =  2
  Number of global     degrees of freedom =    702
  Number of local      degrees of freedom =    696
  Number of prescribed degrees of freedom =      6
  Number of dependent  degrees of freedom =      0


Mass properties of component "gliders"

  Coordinates of reference point:      0.0000,     0.0000,      0.0000

  Rigid body mass matrix:

     3.0907e+02  0.0000e+00 -4.0105e-19  0.0000e+00  3.1935e+01  3.5527e-15
    -1.5407e-33  3.0907e+02  6.0859e-32 -3.1935e+01  3.1393e-32  5.5516e+02
```

```
   -5.8112e-18  3.0815e-33  3.0907e+02 -3.1974e-14 -5.5516e+02  9.8222e-32
    3.8519e-34 -3.1935e+01 -3.5527e-15  2.4698e+03 -1.1369e-13 -1.2979e+02
    3.1935e+01 -2.1185e-32 -5.5516e+02 -5.6843e-14  1.6633e+03  8.8818e-16
   -2.4869e-14  5.5516e+02  2.8696e-32 -1.2979e+02  7.9936e-15  4.0804e+03

 Mass =  3.0907e+02

 Inertia tensor with respect to reference point:

    2.4698e+03 -1.1369e-13 -1.2979e+02
   -5.6843e-14  1.6633e+03  8.8818e-16
   -1.2979e+02  7.9936e-15  4.0804e+03

 Coordinates of center of mass:     1.7962,    -0.0000,     0.1033

 Inertia tensor with respect to center of mass:

    2.4665e+03 -1.7112e-13 -7.2431e+01
   -1.1428e-13  6.6281e+02 -2.4156e-15
   -7.2431e+01  4.6899e-15  3.0832e+03
```

The total mass of the glider including the pilot is $309\ \mathrm{kg}$ which is a realistic value.

3.1.2 Aerodynamic Model and Rigid Trim Analysis

GNU Octave script `rigid.m` defines the aerodynamic model and performs a rigid trim analysis. The aerodynamic component is saved to file `aero.bin` and some of the results to file `rigres.bin`.

Model Definition

The script first loads the parameters from file `params.m` and defines the discretization parameters.

```
# Example: Trim analysis of a standard class glider
#
# b) Rigid Trim
#
# Manoeuvres:
#
#  1. Straight level flight
#  2. Truly banked turn (bank angle = 30°)
#  3. Turn with zero side slip (bank angle = 30°)
#  4. Sudden aileron deflection (deflection angle = 1°)
#
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

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

# ============================================================
# Data
# ============================================================
```

```
# Geometry, Environment and Manoeuvres

  params

# Discretization

  nsa =   5;      % No. of airfoil spline segments
  nxi =  50;      % No. of panels in x-direction of inner wing
  nxo =  40;      % No. of panels in x-direction of outer wing
  nxa =  10;      % No. of panels in x-direction of aileron
  nyi =  20;      % No. of panels in y-direction of inner wing
  nyo =  24;      % No. of panels in y-direction of outer wing
```

Next, the airfoil data are imported and the airfoil is approximated by a piecewise polynomial.

```
# Airfoil: FX 66-S-196 V1

  zs   = dlmread("fx66s196v1.csv", ",", "A100:B143");
  cmbi = mfs_airfoil("fit", "camber", zs, nsa);
  cmbo = mfs_airfoil("fit", "camber", zs, nsa, 0, 1 - fr);
  cmba = mfs_airfoil("fit", "camber", zs, nsa, 1 - fr, 1);
```

The first polynomial is used for the inner wing. It covers the complete airfoil. The second polynomial is used for the front part of the outer wing. It covers the first 80 % of the airfoil. Finally, the third polynomial, covering the last 20 % of the airfoil, is used for the aileron.

Now we define the aerodynamic model. First, we define the points and the lifting surfaces of the right and the left wing. We have to make sure that all lifting surfaces have the same orientation. The identifiers of the lifting surfaces of the right wing begin with 1 and those of the left wing with 2.

In *x*-direction, we use a linear subdivision for the inner wing, the outer wing and the aileron. The values are selected such that the panels have the same length for the outer wing and the aileron.

```
# ============================================================
# Model Definition
# ============================================================

  model = struct("type", "aero", "subtype", "vlm");

  yt  = 0.5 * b;           % y-coordinate of wing tip
  ya  = (1 - lr) * yt;     % y-coordinate of aileron

  dihedral = tand(delta);
  zt = yt * dihedral;      % Wing tip
  za = ya * dihedral;      % Begin of aileron

  ca1 = fr * cr;           % inner aileron chord length
```

```
  ca2 = fr * ct;            % outer aileron chord length
  co1 = cr - ca1;           % inner outer wing chord length
  co2 = ct - ca2;           % outer outer wing chord length

  aa  = (1 - lr) * at;      % Angle of incidence at start of
                            % aileron

# Right wing
# ----------

  points = struct("id",   {1, 2, 3, 4, 5},
                  "coor", {[xle, 0, 0], ...
                           [xle, ya, za], ...
                           [xle, yt, zt], ...
                           [xle + co1, ya, za], ...
                           [xle + co2, yt, zt]});

# Inner wing, outer wing, aileron

  lsf = struct("id",     {10, 11, 12},
               "points", {[1, 2], [2, 3], [4, 5]},
               "chord",  {cr, [co1, co2], [ca1, ca2]},
               "camber", {cmbi, cmbo, cmba},
               "nx",     {nxi, nxo, nxa},
               "ny",     {nyi, nyo, nyo},
               "typex",  "linear",
               "typey",  {"linear", "cos>", "cos>"},
               "alpha",  {[0, aa], [aa, at], [aa, at]});

# ------------------------------------------------------------

# Left wing
# ---------

  points(6 : 10) = struct("id",   {6, 7, 8, 9, 10},
                          "coor", {[xle, 0, 0], ...
                                   [xle, -ya, za], ...
                                   [xle, -yt, zt], ...
                                   [xle + co1, -ya, za], ...
                                   [xle + co2, -yt, zt]});

# Inner wing, outer wing, aileron

  lsf(4 : 6) = struct("id",     {20, 21, 22},
                      "points", {[7, 6], [8, 7], [10, 9]},
                      "chord",  {cr, [co2, co1], [ca2, ca1]},
                      "camber", {cmbi, cmbo, cmba},
                      "nx",     {nxi, nxo, nxa},
                      "ny",     {nyi, nyo, nyo},
                      "typex",  "linear",
                      "typey",  {"linear", "<cos", "<cos"},
                      "alpha",  {[aa, 0], [at, aa], [at, aa]});

# ------------------------------------------------------------
```

We continue with the definition of the points and the lifting surfaces of the vertical and the horizontal stabilizer. The identifiers of the lifting surfaces of the vertical stabilizer begin with 3 and those of the horizontal stabilizer with 4. The orientation of the lifting surfaces of the vertical stabilizer is such that the normal vector points to the left (in opposite *y*-direction).

Again, linear subdivisions are used in *x*-direction in order to achieve at least approximately equal panel lengths.

```
# Vertical stabilizer
# -------------------

  points(11 : 14) = struct("id",   {31, 32, 33, 34},
                           "coor", {[l - 0.6, 0, 0], ...
                                    [l - 0.5, 0, h], ...
                                    [l, 0, 0], ...
                                    [l, 0, h]});
# Fin and rudder

  lsf(7 : 8) = struct("id",     {31, 32},
                      "points", {[31, 32], [33, 34]},
                      "chord",  {[0.6, 0.5], 0.2},
                      "camber", [],
                      "nx",     {30, 10},
                      "ny",     { 8,  8},
                      "typex",  "linear",
                      "typey",  "cos",
                      "alpha",  0);

# ------------------------------------------------------------

# Horizontal stabilizer
# ---------------------

  points(15 : 18) = struct("id",   {41, 42, 43, 44},
                           "coor", {[l - 0.4, 0.5 * be, h], ...
                                    [l, 0.5 * be, h], ...
                                    [l - 0.4, -0.5 * be, h], ...
                                    [l, -0.5 * be, h]});

# Fin

  lsf(9 : 10) = struct("id",     {41, 42},
                       "points", {[32, 41], [43, 32]},
                       "chord",  {[0.5, 0.4], [0.4, 0.5]},
                       "camber", [],
                       "nx",     30,
                       "ny",      8,
                       "typex",  "linear",
                       "typey",  {"cos>", "<cos"},
                       "alpha",  -3);

# Elevator
```

```
  lsf(11 : 12) = struct("id",     {43, 44},
                        "points", {[34, 42], [44, 34]},
                        "chord",  0.1,
                        "camber", [],
                        "nx",     6,
                        "ny",     8,
                        "typex",  "linear",
                        "typey",  {"cos>", "<cos"},
                        "alpha",  -3);

# ------------------------------------------------------------

  model.points = points;
  model.ls     = lsf;
```

The resulting discretization can be seen in Figure 3.1-3. The total number of panels is 5296.

Now, we can define the control surfaces. The following sign conventions are used:

1. A positive value of the aileron means the stick is moved to the right, i.e. the right wing aileron goes up and the left wing aileron goes down.
2. A positive value of the rudder means that the right pedal is pushed, i.e. the rudder moves to the right.

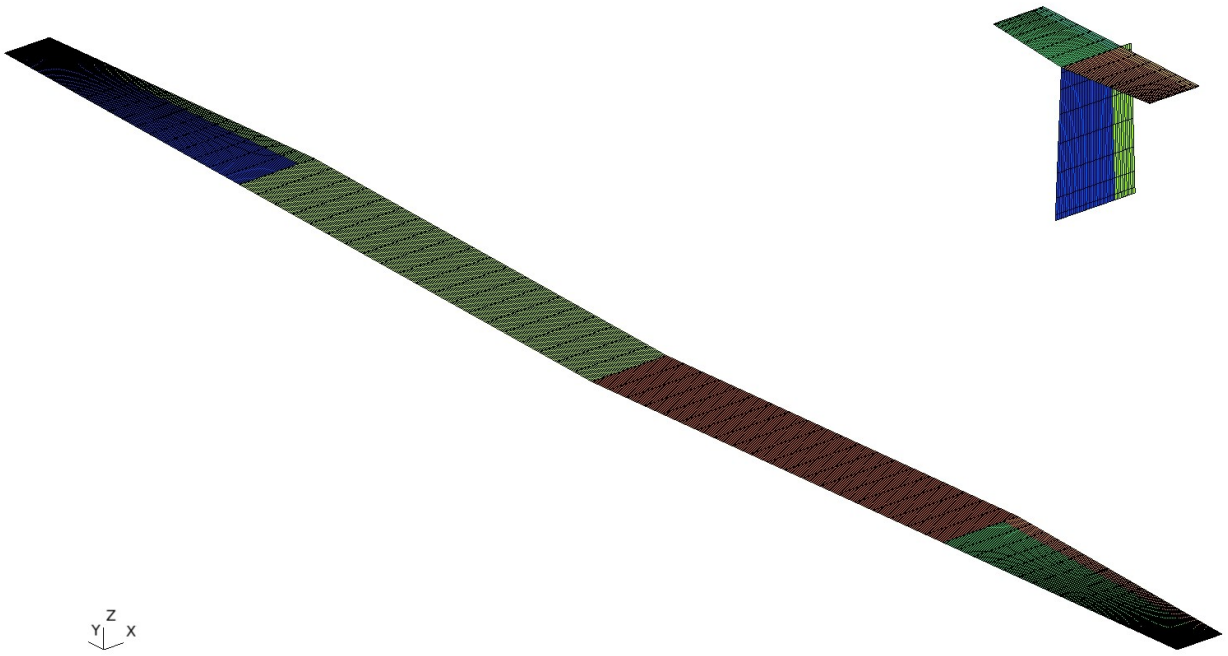


Figure 3.1-3: Aerodynamic Model of the Glider

3. A positive value of the elevator means that the stick is moved back (pulled), i.e. the elevator goes up.

Please note that a positive angle always means a rotation of the control surface towards its negative side, i.e. opposite to its normal vector.

```
# Control surfaces
# ----------------

# Aileron  (positive: stick to the right)
# Rudder   (positive: right pedal)
# Elevator (positive: pull)

  controls = struct("name",    {"aileron", "rudder", "elevator"},
                    "ls",      { [12, 22],    32,     [43, 44]},
                    "factors", { [-1, 1],      1,     [-1, -1]});

  model.controls = controls;
```

The mass properties are taken from the solid model:

```
# Mass properties
# ---------------

  load solid_mass.bin
  model.mass = mp;
```

In a trim analysis, the configurations define the manoeuvres to be analysed. Six of the trim parameters are determined from the dynamic equilibrium conditions. The remaining parameters have to be specified. They define the manoeuvre.

First, we compute the dynamic pressure from the velocity which is the same for all three manoeuvres.

```
# Configurations
# --------------

  qdyn = 0.5 * rho * v^2;
```

The first manoeuvre is a straight level flight. The following trim parameters define the manoeuvre:

1. The linear acceleration **ay** is zero. The linear acceleration **az** equals the gravity acceleration.
2. The angular accelerations **pacce**, **yacce** and **racce** (pitch, yaw and roll accelerations) are zero.
3. The angular velocities **pitch**, **yaw** and **roll** are zero.

The following six trim parameters are determined in the trim analysis:

1. the linear acceleration **ax**
2. the angle of attack **alpha** and the sideslip angle **beta**
3. the three control surface angles **aileron**, **rudder** and **elevator**

Thus, the definition of this manoeuvre is as follows:

```
% Straight level flight

config(1).name  = "Manoeuvre 1: Straight level flight";
config(1).qdyn  = qdyn;
config(1).ay    = 0;
config(1).az    = g;
config(1).pacce = 0;
config(1).yacce = 0;
config(1).racce = 0;
config(1).pitch = 0;
config(1).yaw   = 0;
config(1).roll  = 0;
```

The second manoeuvre is a truly banked turn, see Figure 3.1-4. The vertical acceleration can be computed from the bank angle:

$$g = a_z \cos(\gamma) \;\rightarrow\; a_z = \frac{g}{\cos(\gamma)} \tag{3.1-1}$$

The radius R of the turn can be computed from the bank angle and the flight velocity. The centrifugal force Z_F is

$$Z_F = m\frac{v^2}{R}\ . \tag{3.1-2}$$

Figure 3.1-4 shows that

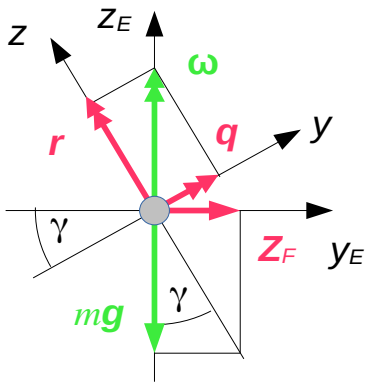


Figure 3.1-4: Truly Banked Turn

$$\tan(\gamma)=\frac{Z_F}{m\,g}=\frac{v^2}{g\,R}\ . \tag{3.1-3}$$

Thus, the radius of the turn is

$$R=\frac{v^2}{g\tan(\gamma)}\ . \tag{3.1-4}$$

The angular velocity ω about the z_E-axis of the earth coordinate system is

$$\omega=\frac{v}{R}=\frac{g}{v}\tan(\gamma) \tag{3.1-5}$$

from which the pitch and yaw rates are obtained:

$$q=\omega\sin(\gamma),\quad r=\omega\cos(\gamma) \tag{3.1-6}$$

For the truly banked turn we have to define the same trim parameters as for the straight level flight, but we have to give them the values computed above. Please note that the pitch and the yaw rate have to be divided by the velocity.

```
% Truly banked turn

tg = tand(gamma);
cs = 1 / sqrt(1 + tg^2);
sn = tg * cs;
w  = tg * g / v;

config(2).name  = "Manoeuvre 2: Truly banked turn";
config(2).qdyn  = qdyn;
config(2).ay    = 0;
config(2).az    = g / cs;
config(2).pacce = 0;
config(2).yacce = 0;
config(2).racce = 0;
config(2).pitch = w * sn / v;
config(2).yaw   = w * cs / v;
config(2).roll  = 0;
```

The six trim parameters that are determined in the trim analysis are the same as in the first manoeuvre.

The third manoeuvre is a left turn as flown by glider pilots. Glider pilots usually try to keep the yaw string parallel to the longitudinal axis of the aircraft, i.e. the side slip angle at the point where the yaw string is attached to the canopy must be zero. The definition of this manoeuvre is a little bit more complicated.

First, as with the truly banked turn, we have (cf. Figure 3.1-4)

$$\omega=\frac{v}{R} \tag{3.1-7}$$

and

$$q = \omega \sin(\gamma), \quad r = \omega \cos(\gamma). \tag{3.1-8}$$

The components of the acceleration with respect to the earth fixed coordinate system must fulfil the conditions

$$a_{y_E} = a_y \cos(\gamma) - a_z \sin(\gamma) = -\frac{v^2}{R} = -\omega v \tag{3.1-9}$$

and

$$a_{z_E} = a_y \sin(\gamma) + a_z \cos(\gamma) = g. \tag{3.1-10}$$

Equation 3.1-10 is already a linear relation between the trim variables a_y and a_z. From the fourth equation we get

$$\omega = -\frac{1}{v}\left(\cos(\gamma) a_y - \sin(\gamma) a_z\right). \tag{3.1-11}$$

Insertion of Equation 3.1-11 into Equations 3.1-8 yields two further linear relations between the trim variables q, r, a_y and a_z:

$$v^2 \frac{q}{v} + \sin(\gamma)\cos(\gamma) a_y - \sin^2(\gamma) a_z = 0 \tag{3.1-12}$$

$$v^2 \frac{r}{v} + \cos^2(\gamma) a_y - \sin(\gamma)\cos(\gamma) a_z = 0 \tag{3.1-13}$$

If the position of the yaw string is a distance d_S in front of the centre of mass and β is the side slip angle at the centre of mass, then the condition that the side slip angle at the position of the yaw string is zero reads

$$\beta + d_S \frac{r}{v} = 0. \tag{3.1-14}$$

Equations 3.1-10 and 3.1-12 to 3.1-14 are four linear relations between the trim variables. In addition, all three angular accelerations and the roll rate are zero. This manoeuvre is thus defined by specifying the values of four trim parameters and four linear relations.

```
% Turn with zero side slip

config(3).name  = "Manoeuvre 3: Turn with zero side slip";
config(3).qdyn  = qdyn;
config(3).pacce = 0;
config(3).yacce = 0;
config(3).racce = 0;
config(3).roll  = 0;

ss = sn * sn; cc = cs * cs; sncs = sn * cs;
vv = v * v;
lincon{1} = struct("ay", sn, "az", cs, "rhs", g);
lincon{2} = struct("ay", sncs, "az", -ss, "pitch", vv);
lincon{3} = struct("ay", cc, "az", -sncs, "yaw", vv);
```

```
lincon{4} = struct("beta", 1, "yaw", ds);

config(3).lincon = lincon;
```

The following ten trim parameters are determined in the trim analysis:

1. the linear accelerations **ax**, **ay** and **az**
2. the angle of attack **alpha** and the sideslip angle **beta** at the centre of mass
3. the pitch rate **pitch** and the yaw rate **yaw**, divided by the velocity
4. the three control surface angles **aileron**, **rudder** and **elevator**

The last manoeuvre is a sudden aileron deflection. This is a quasi-steady manoeuvre, i.e. the results only apply immediately at the start of the manoeuvre.

The following trim parameters define the manoeuvre:

1. The linear acceleration **ay** is zero. The linear acceleration **az** equals the gravity acceleration.
2. The angular accelerations **pacce** and **yacce** (pitch and yaw accelerations) are zero.
3. The angular velocities **pitch**, **yaw** and **roll** are zero.
4. The aileron is deflected.

The following six trim parameters are determined in the trim analysis:

1. the linear acceleration **ax**
2. the angle of attack **alpha** and the sideslip angle **beta**
3. the roll acceleration **racce**
4. the control surface angles **rudder** and **elevator**

Thus, the definition of the manoeuvre is as follows:

```
% Sudden aileron deflection

config(4).name  = "Manoeuvre 4: Sudden aileron deflection";
config(4).qdyn  = qdyn;
config(4).ay    = 0;
config(4).az    = g;
config(4).pacce = 0;
config(4).yacce = 0;
config(4).pitch = 0;
config(4).yaw   = 0;
config(4).roll  = 0;
config(4).aileron = eta;

model.config = config;
```

Analysis

First, we create a new component **glidera** from the model definition and export the mesh for post-processing with Gmsh. In addition, the component is saved to binary file `aero.bin`.

Next, we perform the trim analysis. The trim analysis determines the values of the six trim parameters that are not specified and computes the vortex strengths. The trim parameters are written to the output file.

Subsequently, the panel results, i.e. the panel pressure and forces, are computed and exported to Gmsh for post-processing.

```
# ============================================================
# Analysis
# ============================================================

# Create and export component

  glidera = mfs_new(fid, model);
  mfs_export("aero.msh", "msh", glidera, "mesh");

  save -binary aero.bin gamma v g glidera

# Perform the trim analysis

  glidera = mfs_trim(glidera);
  mfs_print(fid, glidera, "trim", "params");

# Compute and export results

  glidera = mfs_results(glidera, "trim", "panel");
  mfs_export("rigid.pos", "msh", glidera, "trim", "pressure");
```

Next, we retrieve the resulting forces and moments and write them to the output file:

```
# Get load resultants

  [F, M] = mfs_getresp(glidera, "trim", "aeload", mp.cm);
  nconf = columns(F);

  fprintf(fid,
          "\nLoad resultants with respect to center of mass:\n");
  for n = 1 : nconf
      fprintf(fid, "\n  Configuration %2d:\n", n)
      fprintf(fid, "     F = [%10.3e, %10.3e, %10.3e] kN\n",
              F(:, n) / 1000);
      fprintf(fid, "     M = [%10.3e, %10.3e, %10.3e] kNm\n",
              M(:, n) * 1e-3);
```

```
  endfor
```

Then we retrieve the pressure as a function of the *x*-coordinate in two wing sections of the left wing and two wing sections of the right wing and plot it using the GNU Octave plot routines. Two of the sections are approximately in the middle of the inner wing. The other two sections are through the outer wing with the aileron. The positions of the sections are symmetric with respect to the *xz*-plane.

The wing section results are saved to binary file `rigres.bin` so that they can be compared with the results from the flexible trim analysis to be performed later.

```
# Get pressure in some wing sections

  ycmi  = floor(0.5 * nyi); ycmo = floor(0.5 * nyo);
  ycoli = [ycmi, nyi + 1 - ycmi];
  ycolo = [ycmo, nyo + 1 - ycmo];

  [x{1}, p{1}, y(1)] = mfs_xydata(glidera, "trim", "pressure",
                                  [21, 22], ycolo(1));
  [x{2}, p{2}, y(2)] = mfs_xydata(glidera, "trim", "pressure",
                                  20, ycoli(1));
  [x{3}, p{3}, y(3)] = mfs_xydata(glidera, "trim", "pressure",
                                  10, ycoli(2));
  [x{4}, p{4}, y(4)] = mfs_xydata(glidera, "trim", "pressure",
                                  [11, 12], ycolo(2));

  rigres = struct("ycoli", ycoli, "ycolo", ycolo, "p", {p});
  save -binary rigres.bin rigres

  figure(1, "position", [200, 200, 1000, 1000],
            "paperposition", [0, 0, 15, 15]);
  k = 1;
  for n = [1, 4, 2, 3]
     subplot(2, 2, k);
     plot(x{n}, p{n} * 1e-3);
     title(sprintf("y = %6.3f m", y(n)));
     grid;
     if (k == 1)
        legend("Man. 1", "Man. 2", "Man. 3", "Man. 4",
               "location", "south");
     endif
     if (k++ < 3)
        ylim([0, 0.4]);
     else
        ylim([0.1, 0.5]);
        xlabel('x [m]');
     endif
     ylabel('\Delta p [kPa]');
  endfor
  print(["pr", EXT], FORMAT);
```

Figure 3.1-5 shows the resulting diagrams. The upper two diagrams show the pressure in the wing sections containing the aileron and the lower two the pressure in the wing sections of the inner wing.

Finally, we compute the radius of the turn for the second and third manoeuvre and the yaw string angle for all manoeuvres.

For the truly banked turn (configuration 2), the radius of the turn is obtained from Equation 3.1-4. For the turn with zero sideslip angle (configuration 3) we use Equations 3.1-7 and 3.1-11 to get

$$R=\frac{v}{\omega}=\frac{v^2}{\sin(\gamma)a_z-\cos(\gamma)a_y} \quad . \tag{3.1-15}$$

The yaw string angle is calculated according to Equation 3.1-14.

```
# Compute radius of turn and yaw string angle
```

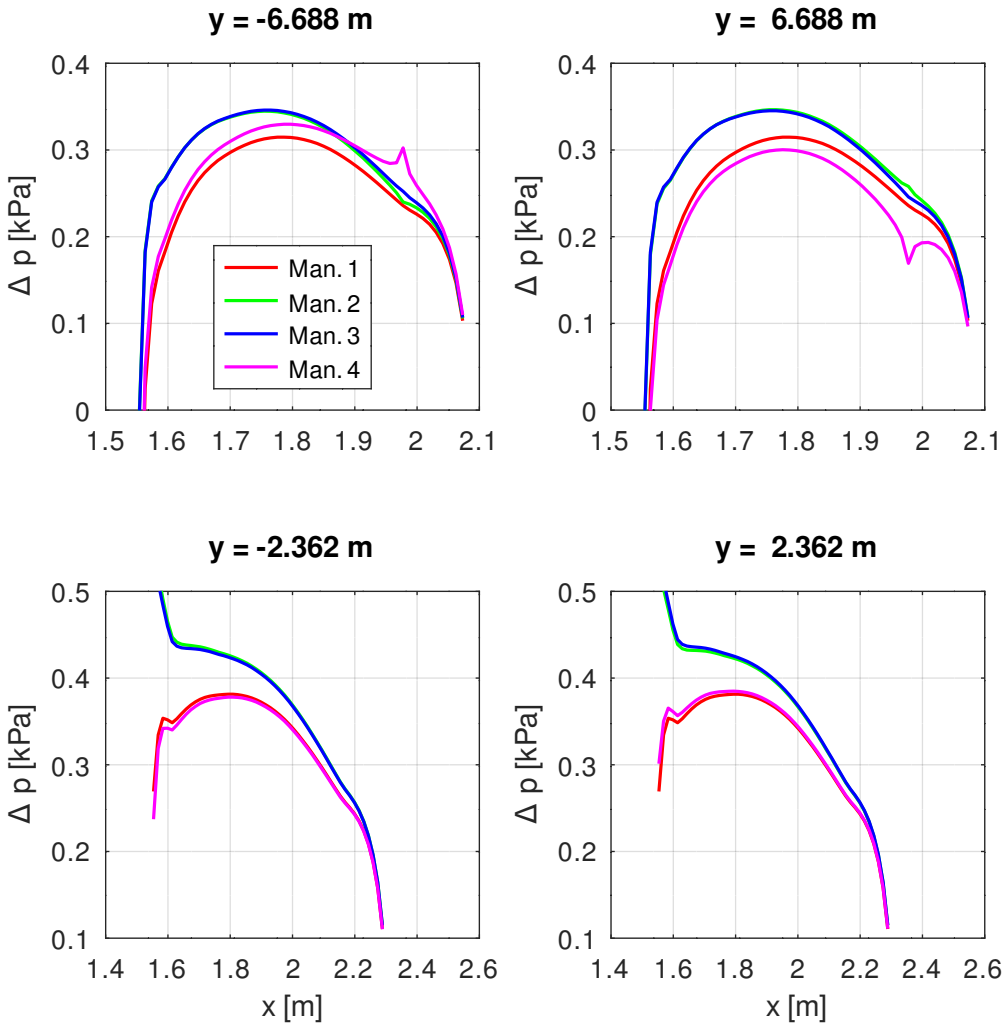


Figure 3.1-5: Pressure from Rigid Trim Analysis in Selected Wing Sections

```
  tp = mfs_getresp(glidera, "trim", "params");

  R2 = vv / (g * tg);
  R3 = vv / (sn * tp.az(3) - cs * tp.ay(3));

  fprintf(fid, "\nRadius of truly banked turn       : %6.2f m\n",
          R2);
  fprintf(fid, "Radius of turn with zero side slip: %6.2f m\n",
          R3);

  ysa = (tp.beta + ds * tp.yaw) * 180 / pi;

  fprintf(fid, "\nYaw string angles:\n");
  fprintf(fid, "  %6.3f, %6.3f, %6.3f, %6.3f deg.\n", ysa);

  fclose(fid);
```

Results

The trim parameters, the load resultants, the radius of the turn as well as the values of the yaw string angle can be found in the output file:

```
Mefisto 2.8: Building new component from input "model"

  Model Type = aero, Model Subtype = vlm

  Number of nodes  =  5804,  Number of panels =  5296
  Number of lifting surfaces =           12
  Number of control surfaces =            3
  Number of configurations   =            4
  Reference chord length     =  0.00000e+00


--------------------------------------------------------------------------------

Component "glidera"

Results of rigid trim analysis (Angles are in degrees)

  Configuration   1: Manoeuvre 1: Straight level flight
  Configuration   2: Manoeuvre 2: Truly banked turn
  Configuration   3: Manoeuvre 3: Turn with zero side slip
  Configuration   4: Manoeuvre 4: Sudden aileron deflection

  Configuration                 1             2             3             4

  qdyn                =   5.4450e+02    5.4450e+02    5.4450e+02    5.4450e+02
  ax                  =   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  ay                  =   0.0000e+00    0.0000e+00   -7.7197e-03    0.0000e+00
  az                  =   9.8100e+00    1.1328e+01    1.1332e+01    9.8100e+00
  racce               =   0.0000e+00    0.0000e+00    0.0000e+00   -1.4195e-01
  pacce               =   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  yacce               =   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  alpha               =   1.8261e+00    2.6072e+00    2.6096e+00    1.8261e+00
  beta                =  -6.2606e-15    1.3212e+00   -3.1275e-01   -2.9329e+00
  pitch               =   0.0000e+00    3.1466e-03    3.1515e-03    0.0000e+00
  yaw                 =   0.0000e+00    5.4500e-03    5.4586e-03    0.0000e+00
  roll                =   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  aileron             =  -8.8535e-15   -1.4043e-01    5.2001e-02    1.0000e+00
  elevator            =  -1.2019e+00    1.2776e+00    1.2828e+00   -1.2019e+00
  rudder              =   6.4431e-15   -4.2086e-01   -2.7640e+00   -4.3191e+00

Load resultants with respect to center of mass:

  Configuration  1:
```

```
     F = [ 0.000e+00,  7.719e-17,  3.032e+00] kN
     M = [-1.590e-14,  2.354e-16,  9.586e-18] kNm

  Configuration  2:
     F = [ 0.000e+00,  1.816e-17,  3.501e+00] kN
     M = [ 1.142e-15,  5.231e-16,  8.931e-17] kNm

  Configuration  3:
     F = [ 0.000e+00, -2.386e-03,  3.502e+00] kN
     M = [ 1.597e-14,  1.022e-15,  5.152e-17] kNm

  Configuration  4:
     F = [ 0.000e+00, -1.265e-17,  3.032e+00] kN
     M = [-3.501e-01,  5.994e-16,  1.028e-02] kNm

Radius of truly banked turn       : 158.90 m
Radius of turn with zero side slip: 158.65 m

Yaw string angles:
  -0.000,  1.633,  0.000, -2.933 deg.
```

Angles are in degrees, and the pitch, yaw and roll rates are rates divided by the flight velocity.

Because there are no forces in *x*-direction, the acceleration **ax** is zero for all three manoeuvres.

Manoeuvres 1 and 4 have the same angle of attack and the same elevator angle. Remember that a positive elevator angle means pull.

As expected, the angle of attack and the elevator angle for the turns are larger than for the straight level flight. The turns need a higher lift, as can be seen from the load resultants.

For the truly banked turn, the sideslip angle **beta** is not zero but its value of 1.32° is small. The yaw string angle has a value of 1.63°. This means that the tip of the yaw string points to the right. For the turn with zero yaw string angle the lateral acceleration **ay** is not zero but its value is small. This explains why it feels more comfortable to fly a turn with the yaw string pointing slightly in the opposite direction. The difference of the radii of the two turns is very small.

For the first two manoeuvres, all load resultants except the force in vertical direction are zero. For the third manoeuvre, there is also a small resulting force in lateral direction.

For the fourth manoeuvre, there is a resulting negative moment about the *x*-axis. This agrees with a negative roll acceleration as is to be expected when the stick is moved to the right. Both this moment and the roll acceleration can be considered a measure for the effectiveness of the aileron.

3.1.3 Aeroelastic Model and Flexible Trim Analysis

GNU Octave script `flexible.m` defines the aerodynamic model and performs a trim analysis of the flexible aircraft. Both a restrained and an unres-

trained trim analysis are performed. The script needs the binary files created in the two preceding steps.

Model Definition

First, we load the solid and aerodynamic components. Next, we define the model type and assign the solid and aerodynamic components to the aeroelastic model. Structures **gliders** and **glidera** can subsequently be deleted to save memory.

```
# Example: Trim analysis of a standard class glider
#
# c) Flexible Trim
#
# Manoeuvres:
#
#  1. Straight level flight
#  2. Truly banked turn (bank angle = 30°)
#  3. Turn with zero side slip (bank angle = 30°)
#  4. Sudden aileron deflection (deflection angle = 1°)
#
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

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

# ============================================================
# Model Definition
# ============================================================

  load solid.bin
  load aero.bin

  model = struct("type", "aeroelastic",
                 "solid", gliders, "aero", glidera);

  clear gliders; clear glidera
```

Next, we define the splines. Lifting surfaces with parallel normal vectors can share the same spline. Thus, there is one spline for the lifting surfaces of the right wing, one for those of the left wing, one for those of the vertical stabilizer and one for those of the horizontal stabilizer.

The spline of the left wing is connected to the nodal points of the left wing and that of the right wing to the nodal points of the right wing.

```
# Splines
# -------
```

```
  nbw = 15;  % Number of wing spline breaks
  nbv =  3;  % Number of vertical stabilizer spline breaks
  nbh =  3;  % Number of horizontal stabilizer spline breaks

  data = struct("nbreaks", {nbw, nbv, nbh});

# Right and left wing splines

  splines(1 : 2) = struct("id",    {10, 20},
                          "type",  "tb",
                          "lsid",  {10 : 12, 20 : 22},
                          "data",  data(1),
                          "nodes", {"right_wing", "left_wing"});
```

The spline of the vertical stabilizer is connected to the nodal points of the vertical stabilizer.

```
# Vertical stabilizer

  splines(3) = struct("id",    30,
                      "type",  "tb",
                      "lsid",  [31, 32],
                      "data",  data(2),
                      "nodes", "vertical");
```

The spline of the horizontal stabilizer is connected to the nodal points of the horizontal stabilizer.

```
# Horizontal stabilizer

  splines(4) = struct("id",   40,
                      "type", "tb",
                      "lsid", [41, 42, 43, 44],
                      "data", data(3),
                      "nodes", "horizontal");

  model.splines = splines;
```

Analysis

First, we create the aeroelastic component and compute the splines.

```
# ============================================================
# Analysis
# ============================================================

# Create aeroelastic component

  glider = mfs_new(fid, model);
```

```
# Compute splines

  glider = mfs_splines(glider);
```

Next, we perform a restrained trim analysis. The input argument to function **mfs_trim** is the structure with the aeroelastic component. Output argument **rtrims** contains the structure with the solid component and output argument **rtrima** that of the aerodynamic component. Aerodynamic results and load resultants are computed as in the previous examples.

```
# Restrained trim analysis
# ------------------------

# Perform the trim analysis

  [rtrims, rtrima] = mfs_trim(glider, "restrained");
  mfs_print(fid, rtrima, "trim", "params");

# Compute and export results

  rtrima = mfs_results(rtrima, "trim", "panel");
  mfs_export("flexible_ra.pos", "msh", rtrima,
             "trim", "pressure", "disp");
  mfs_export("flexible_rs.dsp", "msh", rtrims, "trim", "disp");

# Get load resultants with respect to centre of mass

  load solid_mass.bin
  [Fr, Mr] = mfs_getresp(rtrima, "trim", "aeload", mp.cm);
```

Subsequently, we perform an unrestrained trim analysis. The results are stored in structures **utrims** (solid component) and **utrima** (aerodynamic component).

```
# Unrestrained trim analysis
# --------------------------

# Perform the trim analysis

  [utrims, utrima] = mfs_trim(glider, "unrestrained");
  mfs_print(fid, utrima, "trim", "params");

# Compute and export results

  utrima = mfs_results(utrima, "trim", "panel");
  mfs_export("flexible_ua.pos", "msh", utrima,
             "trim", "pressure", "disp");
  mfs_export("flexible_us.dsp", "msh", utrims, "trim", "disp");

# Get load resultants with respect to centre of mass

  [Fu, Mu] = mfs_getresp(utrima, "trim", "aeload", mp.cm);
```

Next, the load resultants are written to the output file:

```
# Print results
# -------------

  nconf = columns(Fr);

  fprintf(fid,
         "\nLoad resultants with respect to center of mass:\n");

  fprintf(fid, "\n Restrained analysis:\n");
  for n = 1 : nconf
      fprintf(fid, "\n  Configuration %2d:\n", n)
      fprintf(fid, "      F = [%10.3e, %10.3e, %10.3e] kN\n",
              Fr(:, n) / 1000);
      fprintf(fid, "      M = [%10.3e, %10.3e, %10.3e] kNm\n",
              Mr(:, n) * 1e-3);
      mname{n} = ["Man. ", num2str(n)];
  endfor

  fprintf(fid, "\n Unrestrained analysis:\n");
  for n = 1 : nconf
      fprintf(fid, "\n  Configuration %2d:\n", n)
```

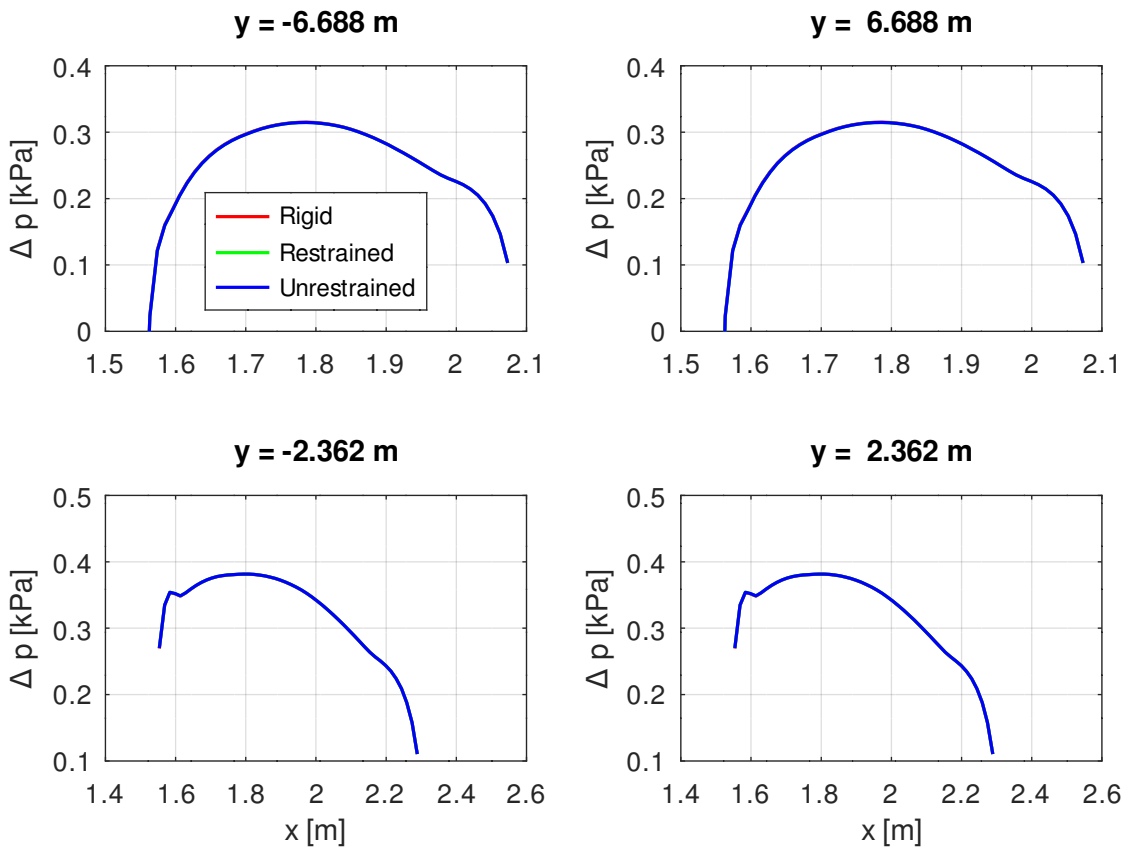


Figure 3.1-6: Manoeuvre 1: Pressure in Selected Wing Sections

```
      fprintf(fid, "     F = [%10.3e, %10.3e, %10.3e] kN\n",
              Fu(:, n) / 1000);
      fprintf(fid, "     M = [%10.3e, %10.3e, %10.3e] kNm\n",
              Mu(:, n) * 1e-3);
  endfor
```

Subsequently, we compare the pressure in the wing sections. A first set of four plots compares the pressures from the rigid trim analysis, the restrained trim analysis and the unrestrained trim analysis for the four different manoeuvres.

```
# Compare pressure in wing sections
# --------------------------------

  load rigres.bin
  ycoli = rigres.ycoli; ycolo = rigres.ycolo; pr = rigres.p;

  [x{1}, pfr{1}, y(1)] = mfs_xydata(rtrima, "trim", "pressure",
                                    [21, 22], ycolo(1));
  [x{2}, pfr{2}, y(2)] = mfs_xydata(rtrima, "trim", "pressure",
                                    20, ycoli(1));
  [x{3}, pfr{3}, y(3)] = mfs_xydata(rtrima, "trim", "pressure",
                                    10, ycoli(2));
```

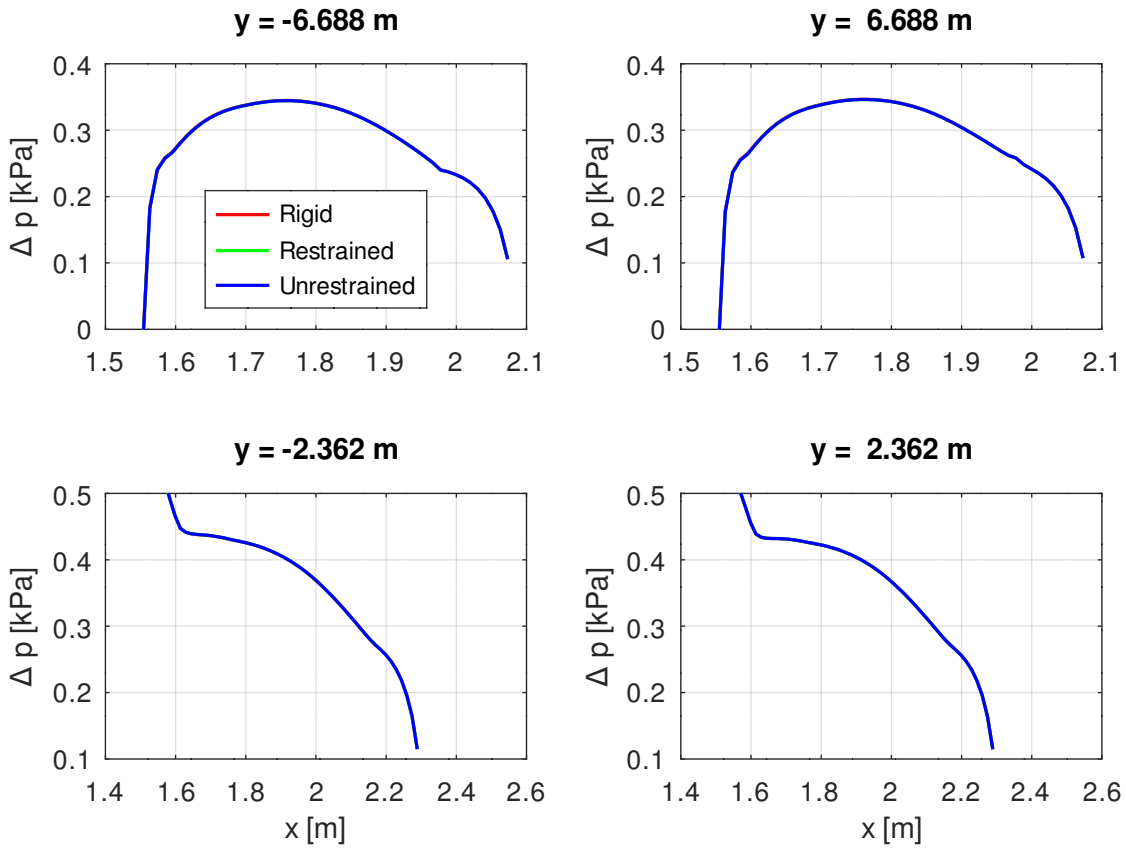


Figure 3.1-7: Manoeuvre 2: Pressure in Selected Wing Sections

```
  [x{4}, pfr{4}, y(4)] = mfs_xydata(rtrima, "trim", "pressure",
                                    [11, 12], ycolo(2));

  [x{1}, pfu{1}, y(1)] = mfs_xydata(utrima, "trim", "pressure",
                                    [21, 22], ycolo(1));
  [x{2}, pfu{2}, y(2)] = mfs_xydata(utrima, "trim", "pressure",
                                    20, ycoli(1));
  [x{3}, pfu{3}, y(3)] = mfs_xydata(utrima, "trim", "pressure",
                                    10, ycoli(2));
  [x{4}, pfu{4}, y(4)] = mfs_xydata(utrima, "trim", "pressure",
                                    [11, 12], ycolo(2));

# Pressure

  for n = 1 : 4
      ytext{n} = sprintf("y = %6.3f m", y(n));
  endfor

  nfig = 1;

  for n = 1 : nconf

      figure(nfig++, "position", [100 + 100 * n, 200, 800, 500],
                     "paperposition", [0, 0, 17, 12]);
      k = 1;
```

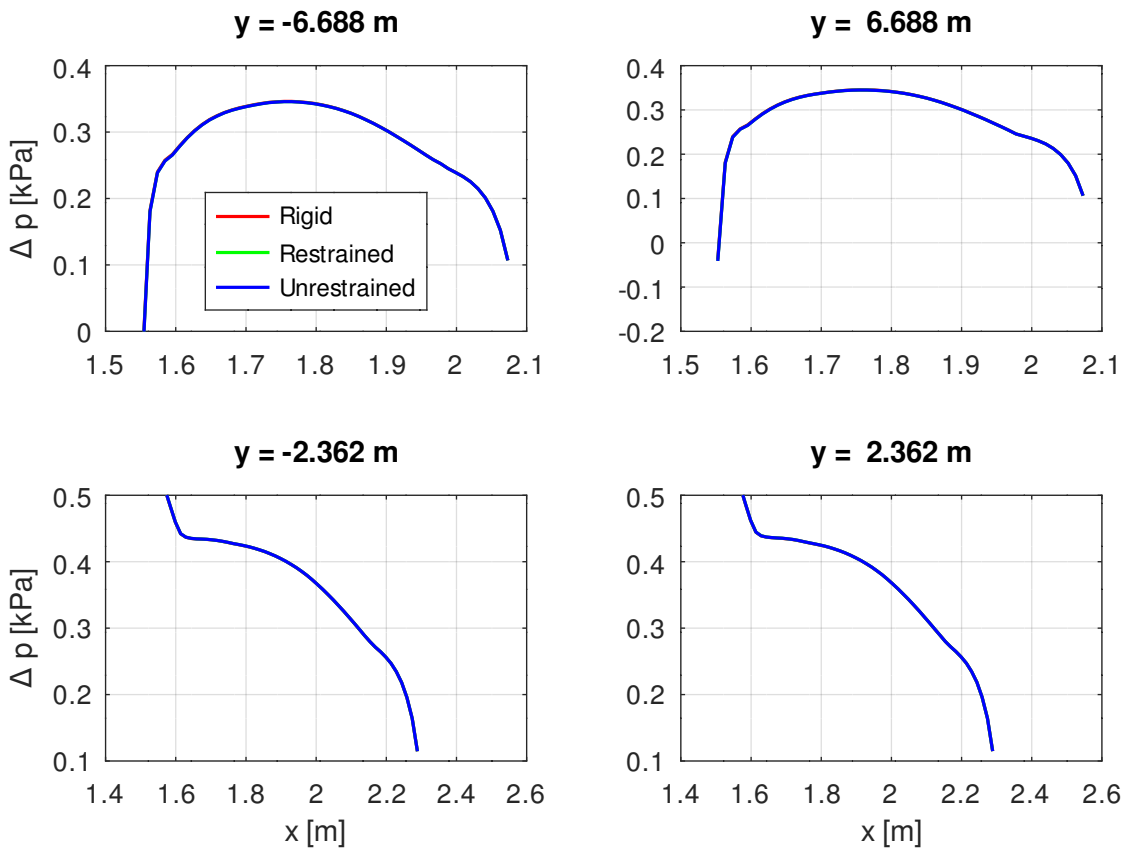


Figure 3.1-8: Manoeuvre 3: Pressure in Selected Wing Sections

```
      for m = [1, 4, 2, 3]
         p = [pr{m}(:, n), pfr{m}(:, n), pfu{m}(:, n)] * 1e-3;
         subplot(2, 2, k);
         plot(x{m}, p);
         title(ytext{m});
         grid;
         if (k == 1)
            legend("Rigid", "Restrained", "Unrestrained",
                   "location", "south");
         endif
         if (k++ < 3)
            ylim([0, 0.4]);
         else
            ylim([0.1, 0.5]);
            xlabel('x [m]');
         endif
         ylabel('\Delta p [kPa]');
      endfor

      print([sprintf("p%d", n), EXT], FORMAT);

   endfor
```

Figures 3.1-6 to 3.1-9 compare the pressure in selected wing sections. No dif-

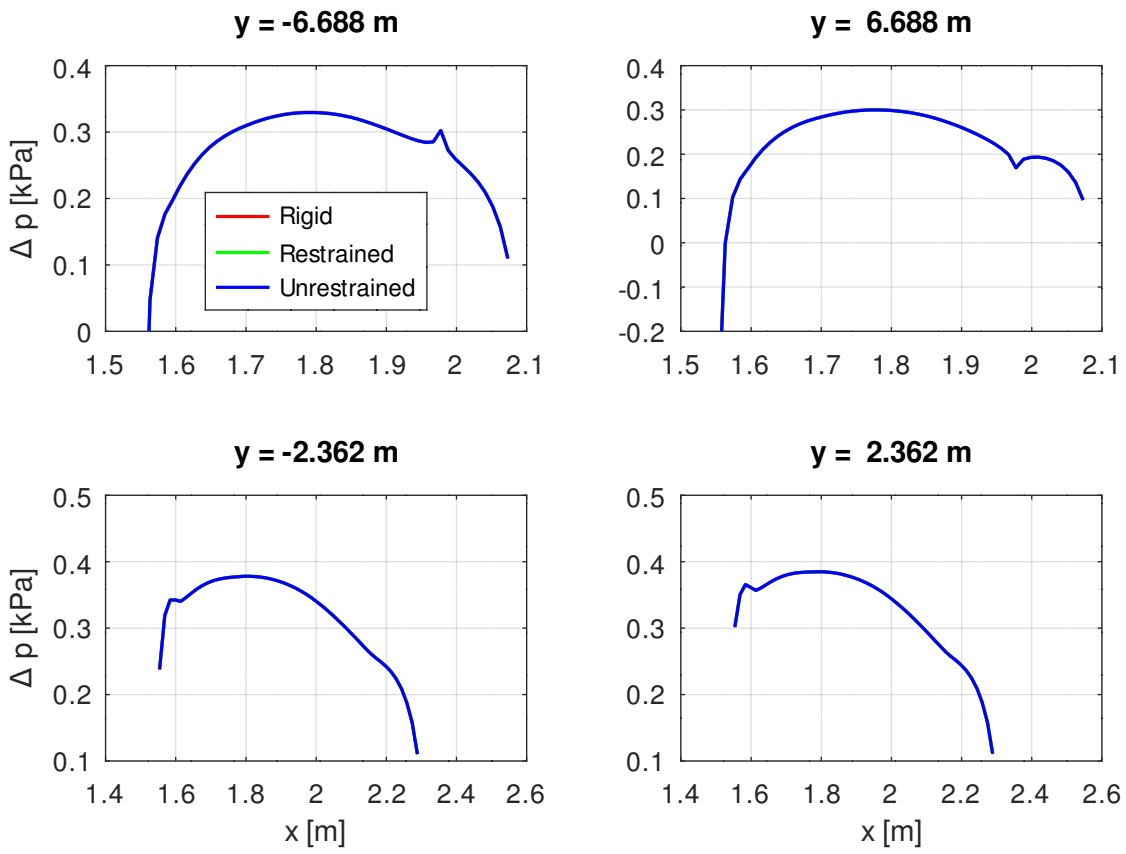


Figure 3.1-9: Manoeuvre 4: Pressure in Selected Wing Sections

ferences can be seen between the rigid trim analysis, the restrained trim analysis and the unrestrained trim analysis. Actually, the pressures from the restrained and the unrestrained trim analysis are identical, but the pressure from the rigid trim analysis is slightly different.

To better see the differences between the rigid and the flexible trim analysis, we produce a plot showing the pressure difference

$$\Delta p = p_r - p_f$$

where p_r is the pressure from the rigid trim analysis and p_f the pressure from the flexible trim analysis. As the restrained and unrestrained analysis should give identical pressure values, we can use either of them.

```
# Difference of pressure (rigid versus flexible)

  figure(nfig++, "position", [200, 400, 700, 500],
                 "paperposition", [0, 0, 17, 12]);
  k = 1;
  for m = [1, 4, 2, 3]
      dp = pr{m} - pfr{m};
      subplot(2, 2, k);
      plot(x{m}, dp);
```

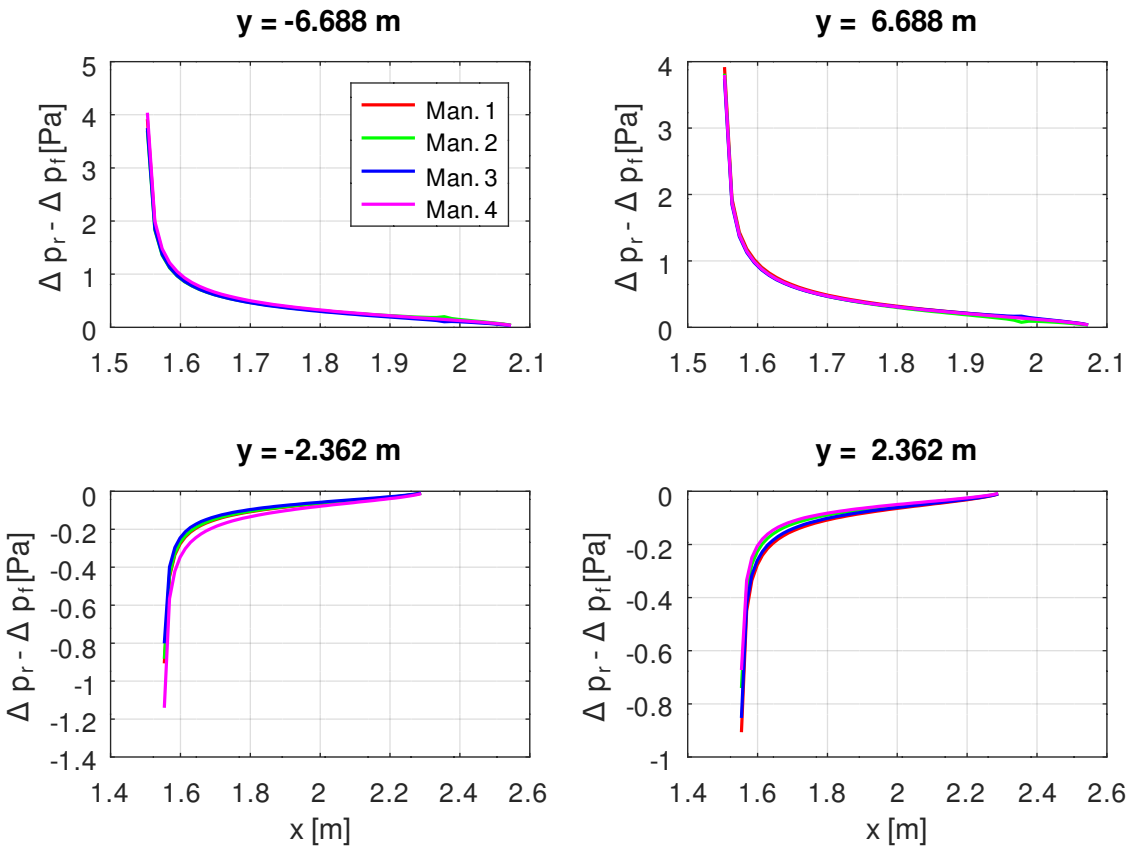


Figure 3.1-10: Pressure Difference between Rigid and Flexible Trim Analysis

```
    title(ytext{m});
    grid;
    if (k == 1)
       legend(mname, "location", "northeast");
    endif
    if (k++ > 2)
       xlabel('x [m]');
    endif
    ylabel('\Delta p_r - \Delta p_f [Pa]');
endfor
print(["dp", EXT], FORMAT);
```

Figure 3.1-10 shows that the difference in pressure between the rigid and the flexible trim analysis is about one hundredth of the size of the pressure.

Next, we examine the distribution of lift in the span direction. Because the differences between the rigid and the flexible aircraft are very small, it is sufficient to look at the results of the flexible aircraft. We can use either the results from the restrained or the unrestrained trim analysis.

```
# Spanwise lift
# -------------

  params

  [s, f, m] = mfs_xydata(utrima, "trim", "spanload",
                         {[21, 22], 20, 10, [11, 12]});
  fz = squeeze(f(3, :, :));
```

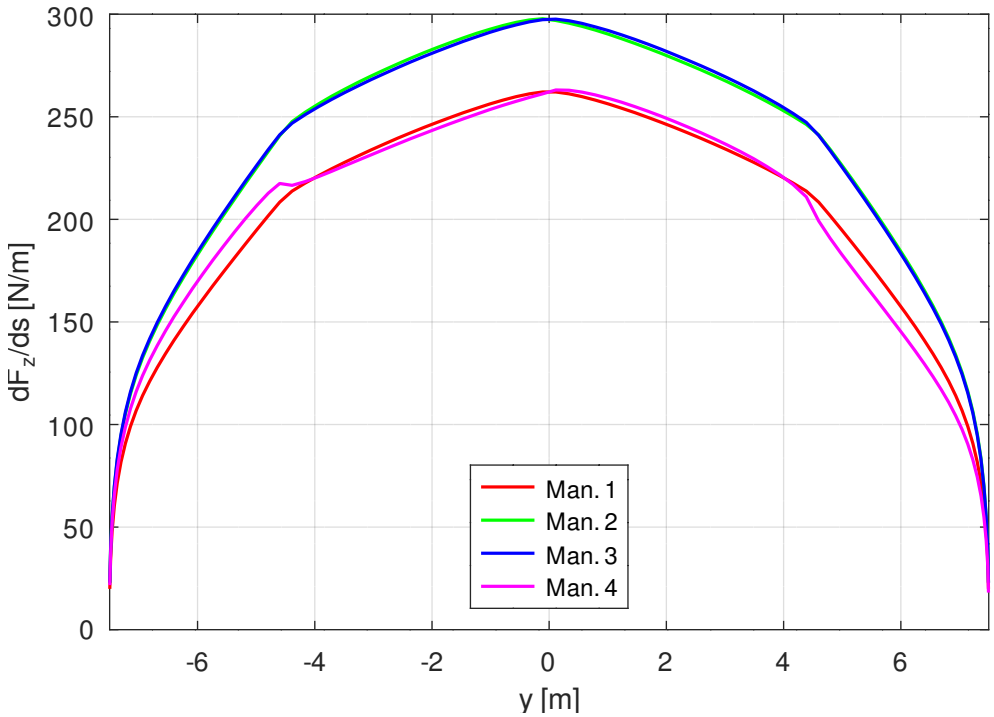


Figure 3.1-11: Spanwise Lift Distribution

```
  y  = s * cosd(delta) - 0.5 * b;

  figure(nfig++, "position", [300, 400, 750, 500],
                 "paperposition", [0, 0, 15, 10]);
  plot(y, fz);
  legend(mname, "location", "south");
  grid;
  xlim(0.5 * [-b, b]);
  xlabel("y [m]"); ylabel("dF_z/ds [N/m]");
  print(["fz", EXT], FORMAT);
```

Figure 3.1-11 shows the lift per length of the wing plotted versus the *y*-coordinate. It can be observed that the loads of the truly banked turn and the turn with zero side slip angle are nearly identical.

The analysis continues with the computation of the stress resultants in the beams. Element results must be identical regardless of whether they are computed from the results of the restrained or the unrestrained analysis. Here we use the results from the unrestrained analysis.

We are particularly interested in the shear force and the bending moment in the main spar. To plot these results along the spar, we first retrieve the element results. The output from function **`mfs_getresp`** is a cell array of structures which we convert into a structure array. This conversion is possible because all elements are beam elements, so all structures have the same fields.

```
# Shear force and bending moment in main spar
# -------------------------------------------

# Get stress resultants and convert results to structure array

  utrims = mfs_results(utrims, "trim", "element");

  Rl = mfs_getresp(utrims, "trim", "resultant",
                   "left_spar", 1 : nconf);
  rl = cell2mat(Rl);

  Rr = mfs_getresp(utrims, "trim", "resultant",
                   "right_spar", 1 : nconf);
  rr = cell2mat(Rr);
```

The structure arrays contain the coordinates of the result points and the stress resultants. First, we create an array with the *y*-coordinates of the result points, from the left wing tip to the right wing tip.

```
# Create array with y-coordinates along spar

  nell = length(Rl); nelr = length(Rr);

  coorl = reshape([rl.coor], 3, nell);
  coorr = reshape([rr.coor], 3, nelr);
```

```
  y     = [flip(coorl(2, :)), coorr(2, :)];
```

Next, we create arrays with the corresponding stress resultants. The stress resultants of interest are the shear force in *z*-direction of the element coordinate system and the bending moment about the element *y*-axis.

```
# Create arrays with stress resultants Qz and My

  Myl = reshape([rl.My], nconf, nell);
  Myr = reshape([rr.My], nconf, nelr);
  My  = [flip(Myl, 2), Myr];

  Qzl = reshape([rl.Qz], nconf, nell);
  Qzr = reshape([rr.Qz], nconf, nelr);
  Qz  = [flip(Qzl, 2), Qzr];
```

Subsequently, the results can be plotted.

```
# Plot the results

  figure(nfig++, "position", [300, 400, 700, 500],
                 "paperposition", [0, 0, 17, 12]);
  subplot(2, 1, 1)
```

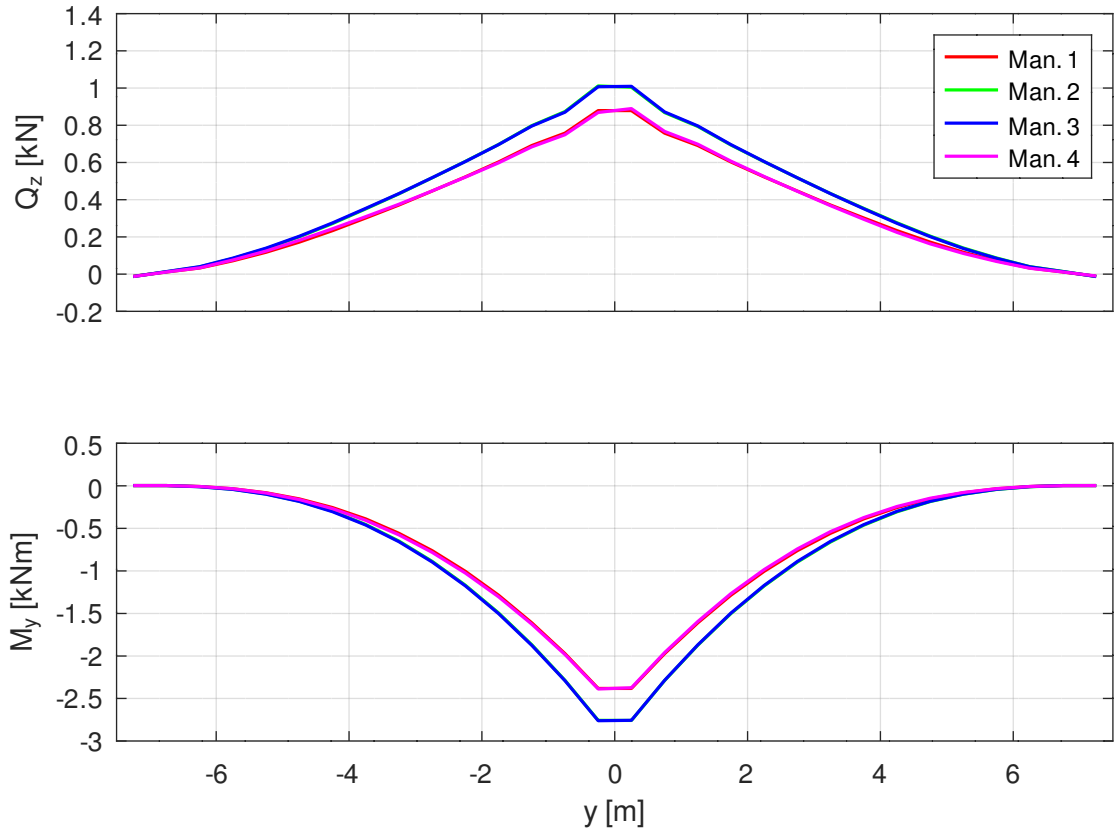


Figure 3.1-12: Shear Force and Bending Moment in Main Spar

```
    plot(y, Qz * 1e-3);
    legend(mname);
    grid;
    xlim([-7.5, 7.5]);
    axis("labely");
    ylabel('Q_z [kN]');
  subplot(2, 1, 2)
    plot(y, My * 1e-3);
    grid;
    xlim([-7.5, 7.5]);
    xlabel('y [m]');
    ylabel('M_y [kNm]');
  print(["QM", EXT], FORMAT);
```

Figure 3.1-12 shows that the shear force and the bending moment are almost identical for manoeuvres 1 and 4 and for manoeuvres 2 and 3 respectively. When interpreting the results, it should be noted that the element *x*-axis of the elements of the left wing points to the left.

Finally, we compute and print the radius of the turn and merge the mesh files and the result files.

```
# Compare radius of turn
# ----------------------

# Truly banked turn

  R2 = v^2 / (g * tand(gamma));

# Turn with zero side slip

  tp(1) = mfs_getresp(rtrima, "trim", "params", 3);
  tp(2) = mfs_getresp(utrima, "trim", "params", 3);

  R3 = v^2 ./ (sind(gamma) * [tp.az] - cosd(gamma) * [tp.ay]);

  fprintf(fid, "\nRadius of turn:\n");
  fprintf(fid, "  Truly banked                 : %6.2f m\n",
               R2);
  fprintf(fid, "  Zero side slip (restrained)  : %6.2f m\n",
               R3(1));
  fprintf(fid, "  Zero side slip (unrestrained): %6.2f m\n",
               R3(2));

# Combine results
# ---------------

  mfs_merge("solid.msh", "aero.msh", "glider.msh", "msh");
  mfs_merge("flexible_rs.dsp", "flexible_ra.pos",
            "glider_r.dsp", "msh");
  mfs_merge("flexible_us.dsp", "flexible_ua.pos",
            "glider_u.dsp", "msh");

  fclose(fid);
```

Results

The output file contains the values of the trim parameters, the load resultants and the radius of the turn. We first look at the trim parameters:

```
Mefisto 2.8: Building new component from input "model"

  Model Type = aeroelastic

  Number of splines      =   4


--------------------------------------------------------------------------------

Component "rtrima"

Results of restrained trim analysis (Angles are in degrees)

  Configuration   1: Manoeuvre 1: Straight level flight
  Configuration   2: Manoeuvre 2: Truly banked turn
  Configuration   3: Manoeuvre 3: Turn with zero side slip
  Configuration   4: Manoeuvre 4: Sudden aileron deflection

  Configuration                1             2             3             4

  qdyn                 =   5.4450e+02    5.4450e+02    5.4450e+02    5.4450e+02
  ax                   =   1.2729e-20    1.4699e-20    1.4704e-20    1.2729e-20
  ay                   =   0.0000e+00    0.0000e+00   -7.8340e-03    0.0000e+00
  az                   =   9.8100e+00    1.1328e+01    1.1332e+01    9.8100e+00
  racce                =   0.0000e+00    0.0000e+00    0.0000e+00   -1.4195e-01
  pacce                =   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  yacce                =   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  alpha                =   1.8853e+00    2.6710e+00    2.6734e+00    1.8853e+00
  beta                 =   1.1120e-03    1.3245e+00   -3.1276e-01   -2.8940e+00
  pitch                =   0.0000e+00    3.1466e-03    3.1516e-03    0.0000e+00
  yaw                  =   0.0000e+00    5.4500e-03    5.4587e-03    0.0000e+00
  roll                 =   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  aileron              =   2.6151e-13   -1.4141e-01    5.2486e-02    1.0000e+00
  elevator             =  -8.2015e-01    1.7044e+00    1.7098e+00   -8.2015e-01
  rudder               =   2.7079e-13   -4.2214e-01   -2.8045e+00   -4.3249e+00

--------------------------------------------------------------------------------

Component "utrima"

Results of unrestrained trim analysis (Angles are in degrees)

  Configuration   1: Manoeuvre 1: Straight level flight
  Configuration   2: Manoeuvre 2: Truly banked turn
  Configuration   3: Manoeuvre 3: Turn with zero side slip
  Configuration   4: Manoeuvre 4: Sudden aileron deflection

  Configuration                1             2             3             4

  qdyn                 =   5.4450e+02    5.4450e+02    5.4450e+02    5.4450e+02
  ax                   =   1.2729e-20    1.4699e-20    1.4704e-20    1.2729e-20
  ay                   =   0.0000e+00    0.0000e+00   -7.8276e-03    0.0000e+00
  az                   =   9.8100e+00    1.1328e+01    1.1332e+01    9.8100e+00
  racce                =   0.0000e+00    0.0000e+00    0.0000e+00   -1.4195e-01
  pacce                =   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  yacce                =   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  alpha                =   1.9374e+00    2.7276e+00    2.7300e+00    1.9374e+00
  beta                 =   1.0737e-13    1.3232e+00   -3.1276e-01   -2.8951e+00
  pitch                =   0.0000e+00    3.1466e-03    3.1516e-03    0.0000e+00
  yaw                  =   0.0000e+00    5.4500e-03    5.4587e-03    0.0000e+00
  roll                 =   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  aileron              =   2.6380e-13   -1.4141e-01    5.2328e-02    1.0000e+00
  elevator             =  -8.2015e-01    1.7044e+00    1.7098e+00   -8.2015e-01
  rudder               =   2.6716e-13   -4.2214e-01   -2.8025e+00   -4.3249e+00

  tx                   =   1.1087e-04    1.2107e-04    1.2110e-04    1.1087e-04
```

```
ty                    =  -8.6864e-06   -1.0157e-05   -7.5551e-06   -4.6381e-06
tz                    =   4.0184e-03    4.7019e-03    4.7039e-03    4.0184e-03
rx                    =   1.7897e-13    4.4598e-04   -3.1394e-04   -1.9351e-03
ry                    =   5.2112e-02    5.6595e-02    5.6607e-02    5.2112e-02
rz                    =   1.1120e-03    1.3037e-03    1.3437e-03    1.1372e-03
```

Except for the turn with zero side slip, the restrained and the unrestrained analysis give the same values of the control surface angles. This is to be expected because these values must not depend on the reference system used. The values of the angle of attack and the sideslip angle, however, are different because the reference axes are different.

The definition of the third manoeuvre, the turn with zero side slip, includes the sideslip angle which depends on the reference axes used. Consequently, the control surface angles from the restrained and the unrestrained analysis are slightly different.

The results of the unrestrained trim analysis include the three translations t_x, t_y, t_z and the three rotation angles r_x, r_y, r_z of the motion of the mean axes system with respect to the body fixed axes. It can be seen that

$$\alpha_u = \alpha_r + r_y$$

where α_u is the angle of attack from the unrestrained analysis and α_r the angle of attack from the restrained analysis.

Table 3.1-1 compares the trim parameter values from the rigid trim analysis with those of the flexible restrained trim analysis. Parameters whose values are all zero are omitted. The angle of attack of the flexible aircraft is larger than that of the rigid aircraft. This indicates that the torsion of the wing decreases the actual angle of attack. The elevator angle of the flexible aircraft is considerably larger than that of the rigid aircraft. This can be an indication

	Manoeuvre 1		Manoeuvre 2		Manoeuvre 4	
	rigid	restrained	rigid	restrained	rigid	restrained
qdyn	$5.445 \cdot 10^2$	$5.445 \cdot 10^2$	$5.445 \cdot 10^2$	$5.445 \cdot 10^2$	$5.445 \cdot 10^2$	$5.445 \cdot 10^2$
az	9.81	9.81	11.33	11.33	9.81	9.81
racce	0	0	0	0	-0.1420	-0.1420
alpha	1.826°	1.885°	2,607°	2,671°	1.826°	1.885°
beta	0	0	1.321°	1.325°	-2,933°	-2,894°
pitch	0	0	$3.1466 \cdot 10^{-3}$	$3.1466 \cdot 10^{-3}$	0	0
yaw	0	0	$5.4500 \cdot 10^{-3}$	$5.4500 \cdot 10^{-3}$	0	0
aileron	0	0	-0.1404°	-0.1414	1°	1°
elevator	-1,202°	-0.8202°	1.278°	1,704°	-1,202°	-0.8202°
rudder	0	0	0.4209°	-0.4221°	-4,319°	-4,325°

Table 3.1-1: Comparison of Trim Parameters

that the bending of the fuselage increases the rigged angle of incidence of the horizontal stabilizer. Figure 3.1-13 confirms this assumption.

For manoeuvre 4, the sudden aileron deflection, the resulting roll acceleration of the flexible aircraft is practically identical with that of the rigid aircraft.

The load resultants read:

```
Load resultants with respect to center of mass:

 Restrained analysis:

  Configuration  1:
     F = [ 0.000e+00,  1.322e-16,  3.032e+00] kN
     M = [-7.155e-17, -2.669e-15,  1.104e-16] kNm

  Configuration  2:
     F = [ 0.000e+00, -2.905e-17,  3.501e+00] kN
     M = [ 4.091e-14, -3.862e-15,  6.490e-17] kNm

  Configuration  3:
     F = [ 0.000e+00, -2.421e-03,  3.502e+00] kN
     M = [ 9.721e-15, -2.593e-15,  2.467e-16] kNm

  Configuration  4:
     F = [ 0.000e+00,  2.880e-17,  3.032e+00] kN
     M = [-3.501e-01, -1.893e-15,  1.028e-02] kNm

 Unrestrained analysis:

  Configuration  1:
     F = [ 0.000e+00,  1.193e-16,  3.032e+00] kN
     M = [ 1.360e-15, -3.200e-15,  7.872e-17] kNm
```

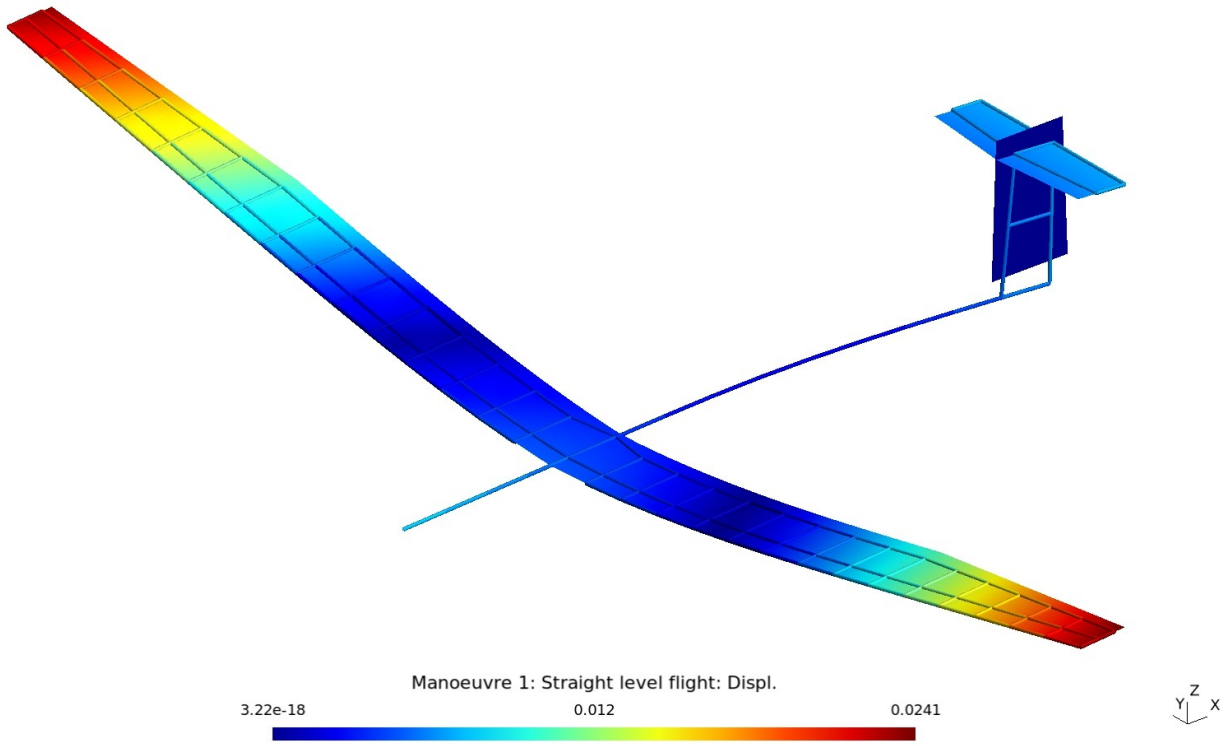


Figure 3.1-13: Manoeuvre 1: Deformation of Solid and Aerodynamic Model (Unrestrained Analysis)

```
  Configuration  2:
     F = [ 0.000e+00, -3.254e-18,  3.501e+00] kN
     M = [ 4.032e-14, -5.068e-15,  8.220e-17] kNm

  Configuration  3:
     F = [ 0.000e+00, -2.419e-03,  3.502e+00] kN
     M = [-1.758e-16, -4.277e-15,  1.049e-16] kNm

  Configuration  4:
     F = [ 0.000e+00, -2.128e-17,  3.032e+00] kN
     M = [-3.501e-01, -1.885e-15,  1.028e-02] kNm
```

The load resultants from the restrained and the unrestrained analysis are identical.

As required by the equilibrium conditions, the resulting forces of the flexible aircraft are identical with those of the rigid aircraft. For manoeuvre 4, the rolling moment M_x of the flexible aircraft is identical with that of the rigid aircraft, indicating an aileron efficiency of 1.

```
Radius of turn:
  Truly banked                 : 158.90 m
  Zero side slip (restrained)  : 158.65 m
  Zero side slip (unrestrained): 158.65 m
```

The values of the radius of the turn are identical with those of the rigid trim analysis. As required, the restrained and the unrestrained trim analysis result in the same radius.

Figure 3.1-13 shows the unrestrained deformation of the solid and the aero-

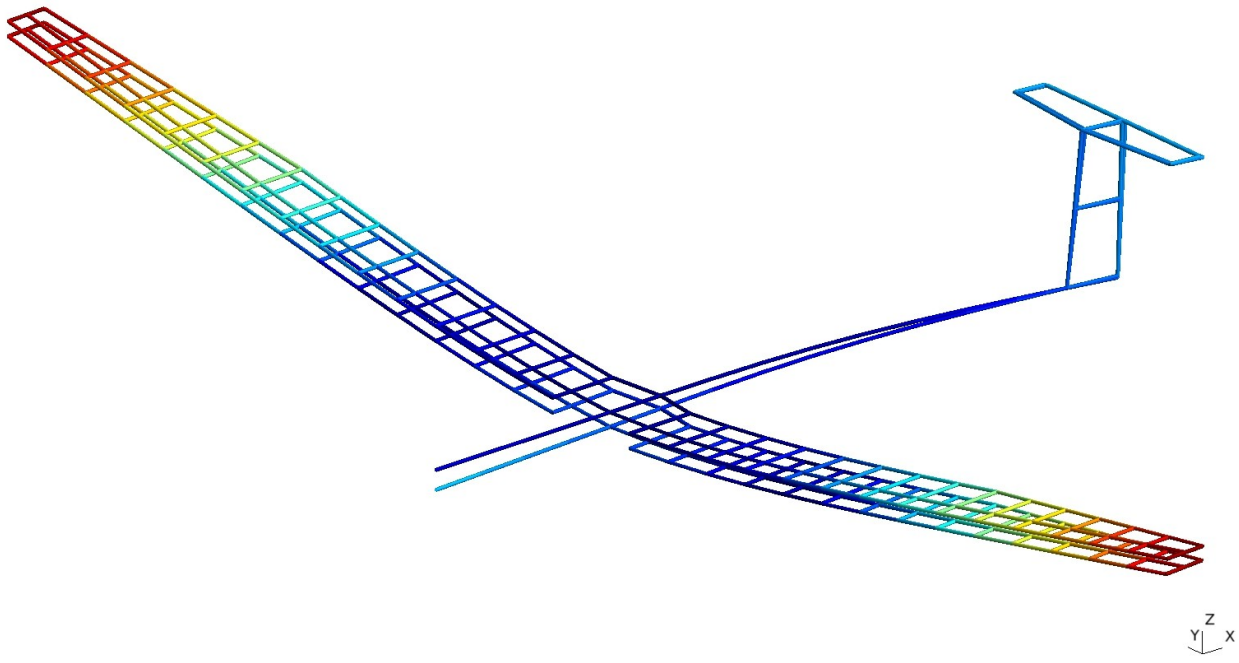


Figure 3.1-14: Manoeuvre 1: Comparison of Restrained and Unrestrained Deformation

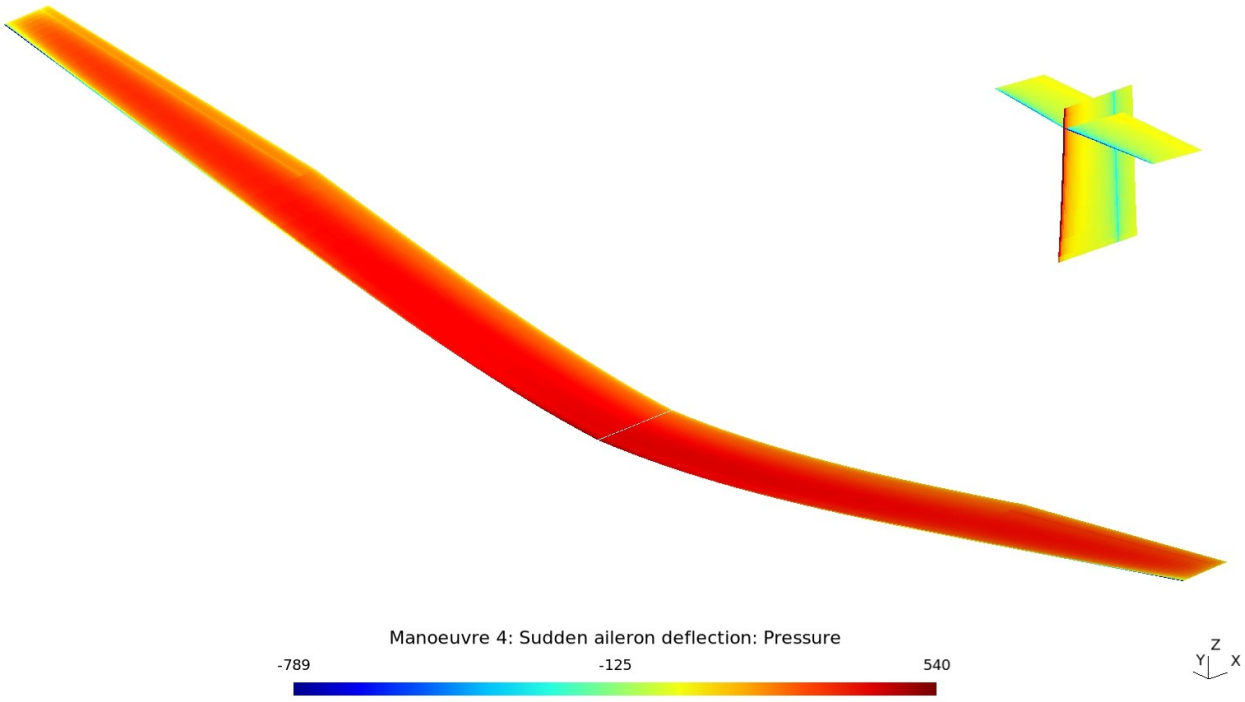


Figure 3.1-15: Manoeuvre 4: Pressure on Deformed Mesh

dynamic model for manoeuvre 1. It can be seen that the deformation normal to the lifting surfaces is identical for both models, confirming a good quality of the splines.

Figure 3.1-14 compares the restrained and the unrestrained deformation for manoeuvre 1. The restrained deformation is relative to a coordinate system that is attached to the solid at the three nodal points with identifiers 6, 101 and 201. The unrestrained deformation is relative to a coordinate system that is attached to the centre of mass.

Figure 3.1-15 shows the pressure on the deformed mesh for manoeuvre 4. The deformation shown is the unrestrained deformation.

4 Flutter Analysis

4.1 Wing

Summary

Directory:	`exa/aeroelastic/flutter/wing`
Objectives:	• learn how to perform a flutter analysis of a wing • learn how to interpret results from a flutter analysis
Elements:	**`b2, m1`**
Method:	Vortex-Lattice
Functions:	**`mfs_beamsection, mfs_import, mfs_new, mfs_stiff, mfs_mass, mfs_massproperties, mfs_freevib, mfs_splines, mfs_transfer, mfs_merge, mfs_flutter, mfs_export, mfs_print, mfs_getresp, mfs_back`**

Problem Description

Perform a flutter analysis of the wing shown in Figure 4.1-1. The velocity range to be examined is from $20\ \mathrm{m/s}$ to $70\ \mathrm{m/s}$. The mass density of the air is $1.21\ \mathrm{kg/m^3}$.

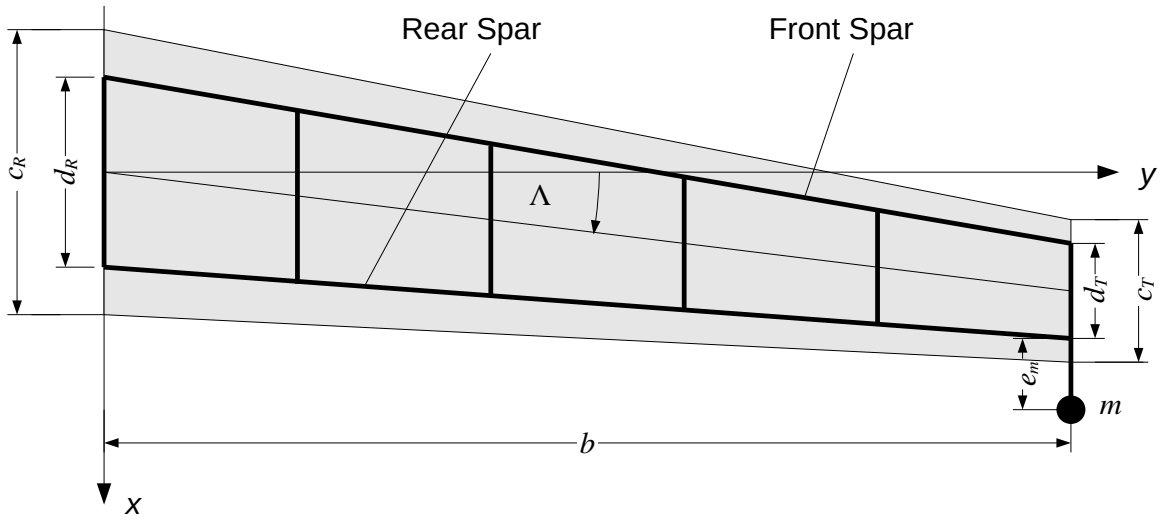
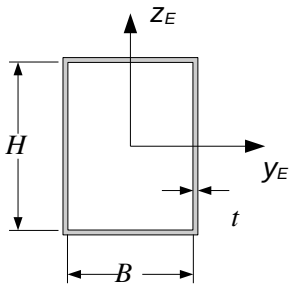


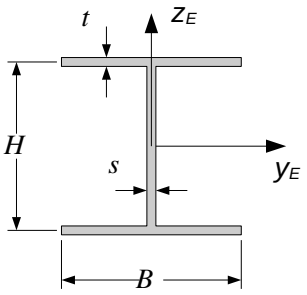
Figure 4.1-1: Geometry of the Wing

Spars:



$B = 200$ mm, $H = 200$ mm
$t = 1$ mm

Ribs:



$B = 100$ mm, $H = 200$ mm
$t = 2$ mm, $s = 2$ mm

Figure 4.1-2: Cross Sections of Spars and Ribs

The solid structure is represented by a beam model, consisting of the front spar, the rear spar and the ribs. In addition, a point mass connected to the wing tip allows to control the flutter speed, cf. Daniella Raveh's flutter demonstator[1] at Technion Israel Institute of Technology.

The dimensions are as follows:

Length of Wing:		$b = 10$ m
Sweep Angle:		$\Lambda = 10°$
Chord Length:	at wing root:	$c_R = 3$ m
	at wing tip:	$c_T = 1.5$ m
Distance of Spars:	at wing root:	$d_R = 2$ m
	at wing tip:	$d_T = 1$ m
Distance of Point Mass:		$e_m = 0.6$ m
Mass of Point Mass:		$m = 5$ kg
Material Data:	Young's modulus:	$E = 70000$ MPa
	Poisson's ratio:	$\nu = 0{,}34$
	Mass density:	$\rho = 2.7$ kg/m^3

The chord length at the wing root is used as reference length for the reduced frequency.

The cross sections of the spars and ribs can be seen in Figure 4.1-2. The cross section of the point mass connector is the same as that of the ribs.

1 https://raveh.net.technion.ac.il/2014/08/14/flutter-demonstrator/

Model Definition

All parameters are defined in file `parameters.m`.

```
# Flutter-Example: Data definition
#
# Units: N, mm, t
#
# If this file is changed, run make_constants.m to update file
# constants.geo which is included by file solid.geo.
#
# ------------------------------------------------------------

# Modal reduction

  nmodes = 15;          % Number of normal modes

# Geometry

  param.sweep =    10;  % Sweep angle in degrees
  param.b     = 10000;  % Half span
  param.dr    =  2000;  % Distance of spars at wing root
  param.dt    =  1000;  % Distance of spars at wing tip
  param.em    =   600;  % Distance of mass from trailing edge
  param.ns    =     5;  % Number of wing sections
  param.el    =   500;  % Element length

  cr          =  3000;  % Chord length at wing root
  ct          =  1500;  % Chord length at wing tip

# Material

  mat = struct("type", "iso",
               "E",   70e3,       % Young's modulus
               "ny",  0.34,       % Poisson's ratio
               "rho", 2.7e-09);   % Mass density

# Mass at wing tip

  mass_geom = struct("m", 5.e-3);

# Cross section properties

  spars = struct("b", 200, "h", 200, "t", 1);
  spars_geom = mfs_beamsection("box", "thin", spars.b, spars.h,
                               spars.t);
  spars_geom.v = [0, 0, 1];

  ribs = struct("b", 100, "h", 200, "t", 2, "s", 2);
  ribs_geom = mfs_beamsection("I", ribs.b, ribs.h, ribs.t,
                              ribs.s);
  ribs_geom.v = [0, 0, 1];

  connector = struct("b", 100, "h", 200, "t", 2, "s", 2);
  connector_geom = mfs_beamsection("I", connector.b, connector.h,
```

```
                                        connector.t, connector.s);
  connector_geom.v = [0, 0, 1];

# Discretization of lifting surface

  nx = 20;  % Number of panels in x-direction
  ny = 45;  % Number of panels in y-direction

# Mass density of air

  rho = 1.21e-12;

# Number of spline breaks

  nbreaks = 10;
```

If you modify this file, don't forget to execute file `make_constants.m` to update file `constants.geo` which is included by file `solid.geo`.

File `modes.m` defines the solid model, creates the solid component and computes the first 15 normal modes. At the end, the solid component is stored in file `solid.bin`.

```
# Flutter Example: Solid model
#
#   - Create the solid component and compute the stiffness
#     and mass matrix
#   - Compute normal modes
#
# ------------------------------------------------------------

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

# Model definition
# ----------------

  parameters

# Translation data

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

  spar = struct("type", "elements", "name", "b2",
                "geom", spars_geom, "mat", mat);
  data.Front_Spar = spar;
  data.Rear_Spar  = spar;

  ribs = struct("type", "elements", "name", "b2",
                "geom", ribs_geom, "mat", mat);
  data.Ribs = ribs;

  connector = struct("type", "elements", "name", "b2",
                     "geom", connector_geom, "mat", mat);
  data.Connector = connector;
```

```
  data.Mass = struct("type", "elements", "name", "m1",
                     "geom", mass_geom);

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

# Analysis
# --------

# Import and translate model

  solid_model = mfs_import(fid, "solid.msh", "msh", data);

# Create and export component

  wings = mfs_new(fid, solid_model);
  mfs_export("solid.axes", "msh", wings, "mesh", "axes");

# Compute Stiffness and mass matrix

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

# Compute Normal modes

  wings = mfs_freevib(wings, nmodes);
  mfs_print(fid, wings, "modes", "freq");
  mfs_export("solid.modes", "msh", wings, "modes", "disp");
  mfs_export("solid.rot", "msh", wings, "modes", "rot");

  save -binary solid.bin wings

  fclose(fid);
```

The solid mesh can be seen in Figure 4.1-4.

File `modes.res` contains the mass properties and the frequencies of the normal modes.

```
Reading model from file "solid.msh", MSH file version 4.1

  Physical Group                    Type
  ------------------------------------------------
  Front_Spar                        elements
  Rear_Spar                         elements
  Ribs                              elements
  Connector                         elements
  Mass                              elements
  Constraints                       constraints

Mefisto 2.8: Building new component from input "solid_model"

  Model Type = solid, Model Subtype = 3d

  Number of nodes      =    80,  Number of elements =    85
  Number of element types =  2
  Number of global     degrees of freedom =    480
  Number of local      degrees of freedom =    468
  Number of prescribed degrees of freedom =     12
```

```
  Number of dependent  degrees of freedom =      0


Mass properties of component "wings"

  Coordinates of reference point:      0.0000,     0.0000,      0.0000

  Rigid body mass matrix:

     6.9447e-02  5.1124e-21  0.0000e+00  0.0000e+00  0.0000e+00 -3.6368e+02
     3.6215e-20  6.9447e-02  0.0000e+00  0.0000e+00  0.0000e+00  7.0372e+01
     0.0000e+00  0.0000e+00  6.9447e-02  3.6368e+02 -7.0372e+01  0.0000e+00
     0.0000e+00  0.0000e+00  3.6368e+02  2.6492e+06 -5.3101e+05  0.0000e+00
     0.0000e+00  0.0000e+00 -7.0372e+01 -5.3101e+05  1.4281e+05  0.0000e+00
    -3.6368e+02  7.0372e+01  0.0000e+00  0.0000e+00  0.0000e+00  2.7913e+06

  Mass =  6.9447e-02

  Inertia tensor with respect to reference point:

     2.6492e+06 -5.3101e+05  0.0000e+00
    -5.3101e+05  1.4281e+05  0.0000e+00
     0.0000e+00  0.0000e+00  2.7913e+06

  Coordinates of center of mass:  1013.3245,  5236.8207,     0.0000

  Inertia tensor with respect to center of mass:

     7.4467e+05 -1.6249e+05  0.0000e+00
    -1.6249e+05  7.1502e+04  0.0000e+00
     0.0000e+00  0.0000e+00  8.1544e+05

--------------------------------------------------------------------------------

Component "wings"

Natural frequencies:

  Mode    Circ. Frequency    Frequency
  -----------------------------------
     1        10.02401       1.59537 Hz
     2        30.34074       4.82888 Hz
     3        55.06064       8.76317 Hz
     4        75.15706      11.96162 Hz
     5        77.89792      12.39784 Hz
     6        83.51369      13.29162 Hz
     7        93.87158      14.94013 Hz
     8        93.94247      14.95141 Hz
     9       107.36703      17.08799 Hz
    10       125.24107      19.93273 Hz
    11       125.27336      19.93787 Hz
    12       138.85065      22.09877 Hz
    13       138.87214      22.10219 Hz
    14       154.29259      24.55643 Hz
    15       163.78747      26.06759 Hz
```

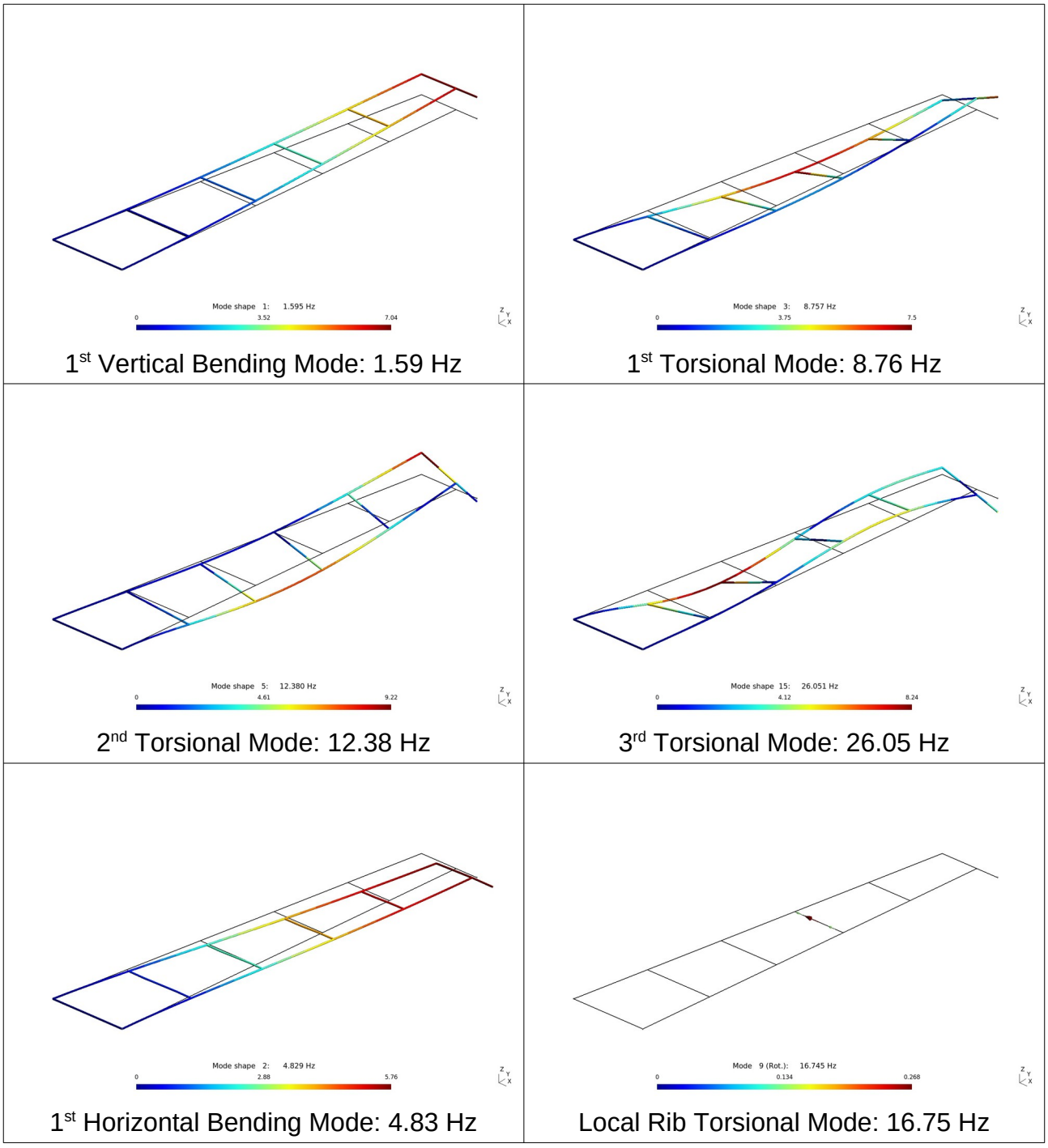
Figure 4.1-3 shows some normal modes. Mode 1 (1st vertical bending), mode 3 (1st torsion), mode 5 (2nd torsion) and mode 15 (3rd torsion) are the most relevant modes for the flutter analysis. Modes 2 and 7 are bending modes in the *xy*-plane that do not influence the aerodynamics. The remaining modes are local torsional modes of the ribs. They can be identified by looking at the rotations (file `solid.rot`).

The aerodynamic model is defined in file `aero.m`. Please note that for time-harmonic aerodynamics, also a reference chord length (field **`cref`**) must be defined. The aerodynamic component is created and stored in file `aero.bin`.

```
# Flutter Example: Aerodynamic model
#
#   - Create the aerodynamic component
#
# ------------------------------------------------------------------

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

# Model definition
```



1st Vertical Bending Mode: 1.59 Hz | 1st Torsional Mode: 8.76 Hz

2nd Torsional Mode: 12.38 Hz | 3rd Torsional Mode: 26.05 Hz

1st Horizontal Bending Mode: 4.83 Hz | Local Rib Torsional Mode: 16.75 Hz

Figure 4.1-3: Some Normal Modes

```
# ----------------

  parameters

  aero_model = struct("type", "aero", "subtype", "vlm",
                      "symy",      0, "cref",       cr);

# Points on leading edge

  xtm = param.b * tand(param.sweep);

  points = struct("id", {1, 2},
                  "coor", {[-0.5 * cr, 0, 0], ...
                           [xtm - 0.5 * ct, param.b, 0]});
  aero_model.points = points;

# Lifting surface

  aero_model.ls = struct("id", 1, "points", [1, 2],
                         "chord", [cr, ct], "nx", nx,
                         "ny", ny, "typey", "cos>");

# Analysis
# --------

# Create and export component

  winga = mfs_new(fid, aero_model);
  mfs_export("aero.msh", "msh", winga, "mesh");

  save -binary aero.bin winga
```

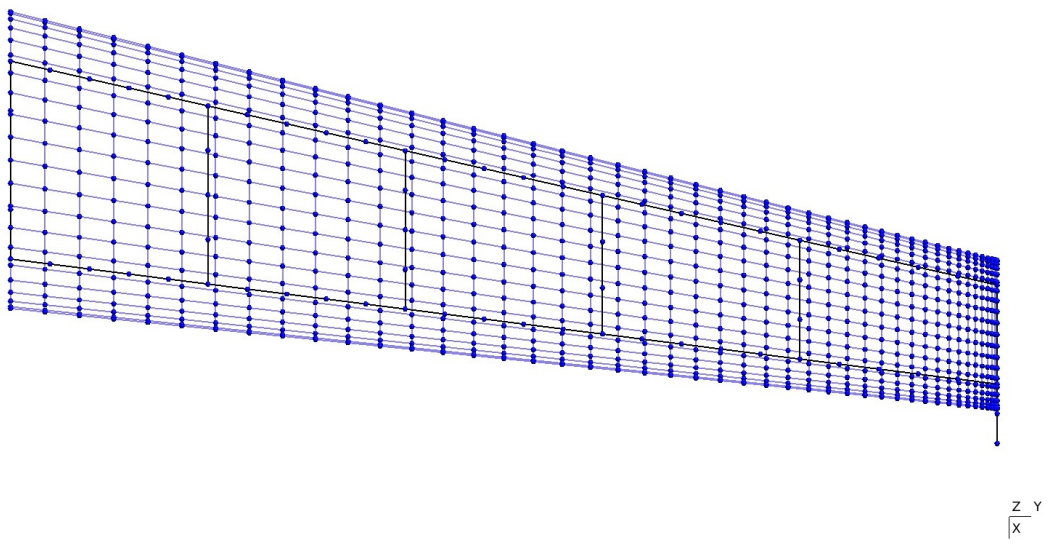


Figure 4.1-4: Solid and Aerodynamic Mesh

```
  fclose(fid);
```

The aerodynamic mesh can be seen in Figure 4.1-4.

Finally, the aeroelastic model is defined in file `aeroelastic.m`. The file contains the definition of the splines. The aeroelastic component is created, the splines are computed and the aeroelastic component is stored in file `aeroelastic.bin`.

```
# Flutter Example: Aeroelastic model
#
#  - Create the aeroelastic component
#  - Check the splines using the normal modes
#
# Files needed:  solid.bin from modes.m
#                aero.bin  from aero.m
#
# ------------------------------------------------------------

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

# Model definition
# ----------------

  parameters

  load solid.bin
  load aero.bin

  aeroelastic_model = struct("type", "aeroelastic",
                             "solid", wings, "aero", winga);

# Splines

  spldata = struct("nbreaks", nbreaks);
  aeroelastic_model.splines = struct("id", 1, "type", "tb",
                                     "lsid", 1, "data", spldata);

# Analysis
# --------

  wing = mfs_new(fid, aeroelastic_model);
  wing = mfs_splines(wing);

  save -binary aeroelastic.bin wing
```

To check the quality of the splines, the normal modes are transferred to the aerodynamic model and exported to Gmsh. Finally, the solid and the aerodynamic mesh are combined into file `wing.msh`. Also, the files with the normal modes are combined into one file `wing.modes`. These files can be used to visualize the deformations of the solid and the aerodynamic mesh within the same plot.

```
winga = mfs_transfer(wing, wings, "modes", "disp");
mfs_export("aero.modes", "msh", winga, "modes", "disp");

mfs_merge("solid.msh", "aero.msh", "wing.msh", "msh");
mfs_merge("solid.modes", "aero.modes", "wing.modes", "msh");

fclose(fid);
```

Figure 4.1-4 shows the solid and the aerodynamic mesh. It can be seen that the geometry of both models has been defined correctly.

Figure 4.1-5 shows the deformation of the aerodynamic mesh for mode 15, together with the deformation of the solid mesh. It can be seen that the deformations of the two meshes match very well.

Flutter Analysis

First, we use the k-method to search the velocity range of interest for possible flutter velocities. The k-method is the fastest method but its results are sometimes difficult to interpret.

From the results of the normal modes analysis, we know that the lowest circular frequency is about $10\ \mathrm{s^{-1}}$ and the largest circular frequency computed is about $164\ \mathrm{s^{-1}}$. From this, we compute

$$k_{min} = \frac{\omega_{min}\, c_{ref}}{2\, v_{max}} = \frac{10 \cdot 3\ \mathrm{m/s}}{2 \cdot 70\ \mathrm{m/s}} = 0.2\,, \quad k_{max} = \frac{\omega_{max}\, c_{ref}}{2\, v_{min}} = \frac{164 \cdot 3}{2 \cdot 20} = 12.3 \quad .$$

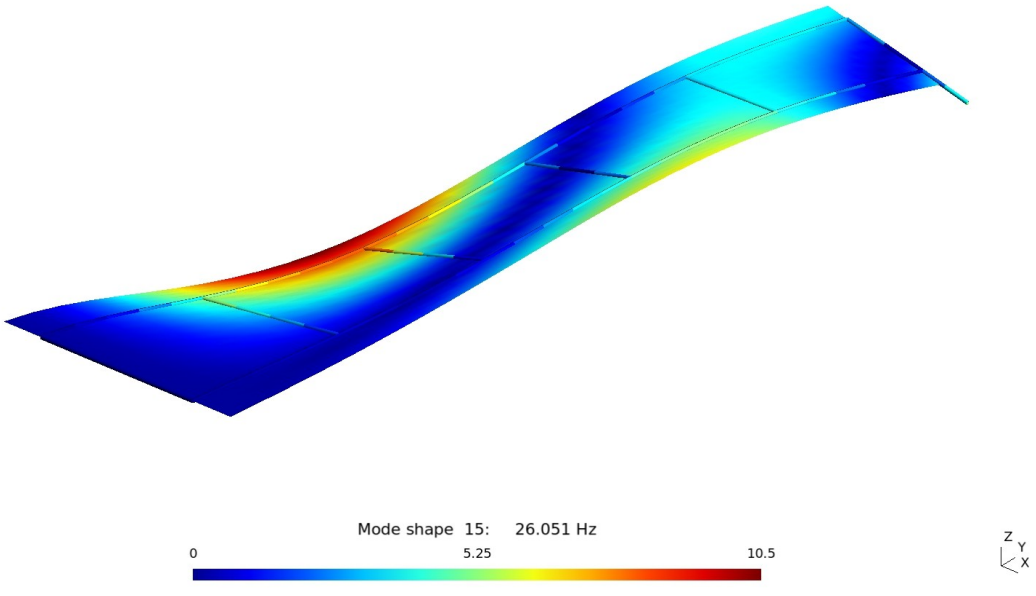


Figure 4.1-5: Mode 15: Solid and Aerodynamic Model

As the modal basis ensures good results only up to $\omega_c \approx 0.3\,\omega_{max}$, an initial analysis is performed with reduced frequencies from 0.5 to 4.0 with increments of 0.5. File `flutter_k1.m` executes this flutter analysis, stores the component in file `flutter_k1.bin` for further postprocessing and prints the results to the output file.

```
# Flutter Example: Flutter analysis using the k-method
#                  Initial analysis using coarse set of reduced
#                  frequencies
#
# File needed: aeroelastic.bin from aeroelastic.m
#
# ------------------------------------------------------------------

  fid  = fopen("flutter_k1.res", "wt");

# Data

  parameters

  kred = 0.5 : 0.5 : 4.0;   % Reduced frequencies

# Load aeroelastic component

  load aeroelastic.bin

# Flutter analysis

  wing = mfs_flutter(wing, kred, rho, "k");
  save -binary flutter_k1.bin wing

  mfs_print(fid, wing, "flutter", "curves");

  fclose(fid);
```

To decide which curves to select for plotting, we look at the output file.

```
--------------------------------------------------------------------------------

Component "wing"

Results of flutter analysis, k-method

 Flutter mode  1

 Point:      k           v           g           f
 ------------------------------------------------------
     1:    0.500000  2.8833e+04  -1.144e+00     1.52965
     2:    1.000000  1.3719e+04  -4.537e-01     1.45560
     3:    1.500000  8.9789e+03  -2.753e-01     1.42903
     4:    2.000000  6.6795e+03  -1.970e-01     1.41743
     5:    2.500000  5.3205e+03  -1.534e-01     1.41130
     6:    3.000000  4.4225e+03  -1.257e-01     1.40774
     7:    3.500000  3.7846e+03  -1.065e-01     1.40546
     8:    4.000000  3.3077e+03  -9.232e-02     1.40384

 Flutter mode  2

 Point:      k           v           g           f
```

Point:	k	v	g	f
1:	0.500000	8.5115e+04	2.995e-01	4.51550
2:	1.000000	5.6000e+04	-4.542e-02	5.94178
3:	1.500000	4.2205e+04	-8.173e-02	6.71712
4:	2.000000	3.3320e+04	-7.240e-02	7.07076
5:	2.500000	2.7321e+04	-6.033e-02	7.24722
6:	3.000000	2.3077e+04	-5.079e-02	7.34562
7:	3.500000	1.9942e+04	-4.352e-02	7.40563
8:	4.000000	1.7541e+04	-3.793e-02	7.44450

Flutter mode 3

Point:	k	v	g	f
1:	0.500000	1.7431e+05	-6.342e-01	9.24730
2:	1.000000	9.9039e+04	-4.230e-01	10.50833
3:	1.500000	6.6739e+04	-3.086e-01	10.62185
4:	2.000000	5.0225e+04	-2.456e-01	10.65806
5:	2.500000	4.0256e+04	-2.037e-01	10.67826
6:	3.000000	3.3586e+04	-1.736e-01	10.69077
7:	3.500000	2.8810e+04	-1.508e-01	10.69877
8:	4.000000	2.5220e+04	-1.331e-01	10.70362

Flutter mode 4

Point:	k	v	g	f
1:	0.500000	4.6291e+05	-2.213e-05	24.55799
2:	1.000000	2.3145e+05	-2.795e-05	24.55777
3:	1.500000	1.5430e+05	-2.730e-05	24.55772
4:	2.000000	1.1573e+05	-2.567e-05	24.55771
5:	2.500000	9.2580e+04	-2.380e-05	24.55772
6:	3.000000	7.7150e+04	-2.193e-05	24.55773
7:	3.500000	6.6129e+04	-2.017e-05	24.55775
8:	4.000000	5.7863e+04	-1.857e-05	24.55776

Flutter mode 5

Point:	k	v	g	f
1:	0.500000	4.1664e+05	-1.406e-04	22.10359
2:	1.000000	2.0831e+05	-2.503e-04	22.10203
3:	1.500000	1.3887e+05	-3.647e-04	22.10161
4:	2.000000	1.0415e+05	-4.946e-04	22.10125
5:	2.500000	8.3318e+04	-6.289e-04	22.10075
6:	3.000000	6.9430e+04	-7.601e-04	22.10010
7:	3.500000	5.9509e+04	-8.842e-04	22.09929
8:	4.000000	5.2068e+04	-9.990e-04	22.09836

Flutter mode 6

Point:	k	v	g	f
1:	0.500000	2.5051e+05	-5.304e-05	13.29007
2:	1.000000	1.2526e+05	-7.051e-05	13.29027
3:	1.500000	8.3506e+04	-5.718e-05	13.29039
4:	2.000000	6.2630e+04	-4.587e-05	13.29043
5:	2.500000	5.0104e+04	-3.801e-05	13.29046
6:	3.000000	4.1753e+04	-3.239e-05	13.29047
7:	3.500000	3.5789e+04	-2.818e-05	13.29048
8:	4.000000	3.1315e+04	-2.489e-05	13.29048

Flutter mode 7

Point:	k	v	g	f
1:	0.500000	2.8180e+05	-5.875e-05	14.95018
2:	1.000000	1.4091e+05	-1.822e-05	14.95098
3:	1.500000	9.3941e+04	-9.989e-06	14.95112
4:	2.000000	7.0456e+04	-6.654e-06	14.95117
5:	2.500000	5.6365e+04	-4.956e-06	14.95119
6:	3.000000	4.6971e+04	-3.951e-06	14.95120
7:	3.500000	4.0261e+04	-3.287e-06	14.95120
8:	4.000000	3.5228e+04	-2.816e-06	14.95121

```
Flutter mode  8

Point:      k           v           g           f
-----------------------------------------------------
    1:    0.500000  3.2789e+05  -1.054e-02    17.39492
    2:    1.000000  1.8353e+05  -9.116e-02    19.47319
    3:    1.500000  1.3024e+05  -1.000e-01    20.72864
    4:    2.000000  1.0055e+05  -8.917e-02    21.33702
    5:    2.500000  8.1651e+04  -7.706e-02    21.65854
    6:    3.000000  6.8629e+04  -6.682e-02    21.84519
    7:    3.500000  5.9140e+04  -5.856e-02    21.96244
    8:    4.000000  5.1933e+04  -5.187e-02    22.04091

Flutter mode  9

Point:      k           v           g           f
-----------------------------------------------------
    1:    0.500000  3.2210e+05   2.878e-04    17.08790
    2:    1.000000  1.6104e+05  -3.790e-05    17.08726
    3:    1.500000  1.0736e+05  -4.226e-05    17.08730
    4:    2.000000  8.0522e+04  -3.918e-05    17.08735
    5:    2.500000  6.4418e+04  -3.509e-05    17.08739
    6:    3.000000  5.3682e+04  -3.125e-05    17.08742
    7:    3.500000  4.6013e+04  -2.792e-05    17.08744
    8:    4.000000  4.0261e+04  -2.511e-05    17.08746

Flutter mode 10

Point:      k           v           g           f
-----------------------------------------------------
    1:    0.500000  3.7601e+05  -2.325e-04    19.94785
    2:    1.000000  1.8796e+05  -1.587e-03    19.94304
    3:    1.500000  1.2522e+05  -8.913e-04    19.92872
    4:    2.000000  9.3912e+04  -4.177e-04    19.92880
    5:    2.500000  7.5131e+04  -2.601e-04    19.92921
    6:    3.000000  6.2610e+04  -1.879e-04    19.92946
    7:    3.500000  5.3666e+04  -1.475e-04    19.92960
    8:    4.000000  4.6958e+04  -1.217e-04    19.92969

Flutter mode 11

Point:      k           v           g           f
-----------------------------------------------------
    1:    0.500000  3.7574e+05   1.088e-05    19.93370
    2:    1.000000  1.8787e+05  -1.389e-05    19.93405
    3:    1.500000  1.2525e+05  -3.860e-05    19.93443
    4:    2.000000  9.3940e+04  -4.808e-05    19.93475
    5:    2.500000  7.5153e+04  -4.698e-05    19.93497
    6:    3.000000  6.2628e+04  -4.287e-05    19.93511
    7:    3.500000  5.3681e+04  -3.851e-05    19.93519
    8:    4.000000  4.6971e+04  -3.461e-05    19.93525
```

The velocities in this listing are in $\mathrm{mm/s}$. Please note that flutter modes are not sorted according to ascending frequencies.

For flutter mode 2, the damping parameter γ (column **g**) has a zero crossing between the reduced frequencies 0.5 and 1.0, corresponding to velocities of $85.1\ \mathrm{m/s}$ and $56{,}0\ \mathrm{m/s}$, respectively, indicating that this flutter mode becomes unstable within the velocity range of interest.

Next, we plot the flutter curves of the first five flutter modes with the lowest frequencies. These are the modes 1 to 3, 6 and 7. The plots are created by GNU Octave script `plot_k1.m`.

```
# Flutter Example: Postprocessing of results from k-method
#
```

```
# - plot flutter curves of first analysis
#
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

  modes = [1, 2, 3, 6, 7];      % selected flutter modes

  load flutter_k1.bin

  [v, g, kred, f] = mfs_getresp(wing, "flutter", "curves", modes);
  v = v / 1000;

  nofmod = length(modes);

  for n = 1 : nofmod
      ltext{n} = sprintf("Mode %2d", modes(n));
  endfor

  figure(1, "position", [50, 50, 700, 500],
            "paperposition", [0, 0, 17, 11]);
  subplot(2, 1, 1)
     plot(v, g, "marker", "o");
     legend(ltext, "location", "southeast");
     legend("numcolumns", 2);
     grid;
```

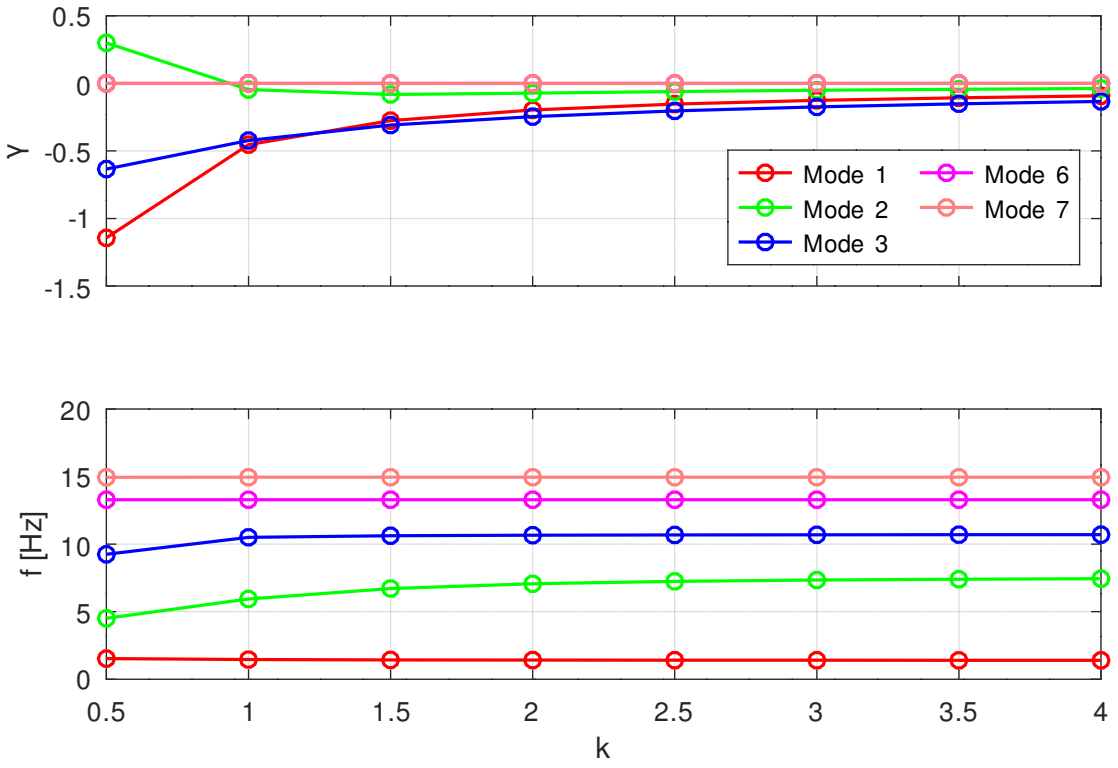


Figure 4.1-6: Flutter Results versus Reduced Frequency (k-method)

```
    axis("labely");
    xlim([0, 100]);
    ylabel('\gamma');
subplot(2, 1, 2)
    plot(v, f, "marker", "o");
    grid;
    xlim([0, 100]);
    xlabel("v [m/s]"); ylabel("f [Hz]");
print(["flutter_k1_v", EXT], FORMAT);

figure(2, "position", [200,100, 700, 500],
          "paperposition", [0, 0, 17, 11]);
subplot(2, 1, 1)
    plot(kred, g, "marker", "o");
    legend(ltext, "location", "southeast");
    legend("numcolumns", 2);
    grid;
    axis("labely");
    ylabel('\gamma');
subplot(2, 1, 2)
    plot(kred, f, "marker", "o");
    grid;
    xlabel("k"); ylabel("f [Hz]");
print(["flutter_k1_k", EXT], FORMAT);
```

Figure 4.1-6 shows the damping parameter and the frequency plotted versus the reduced frequency. Mode 2 shows a zero crossing of the damping para-

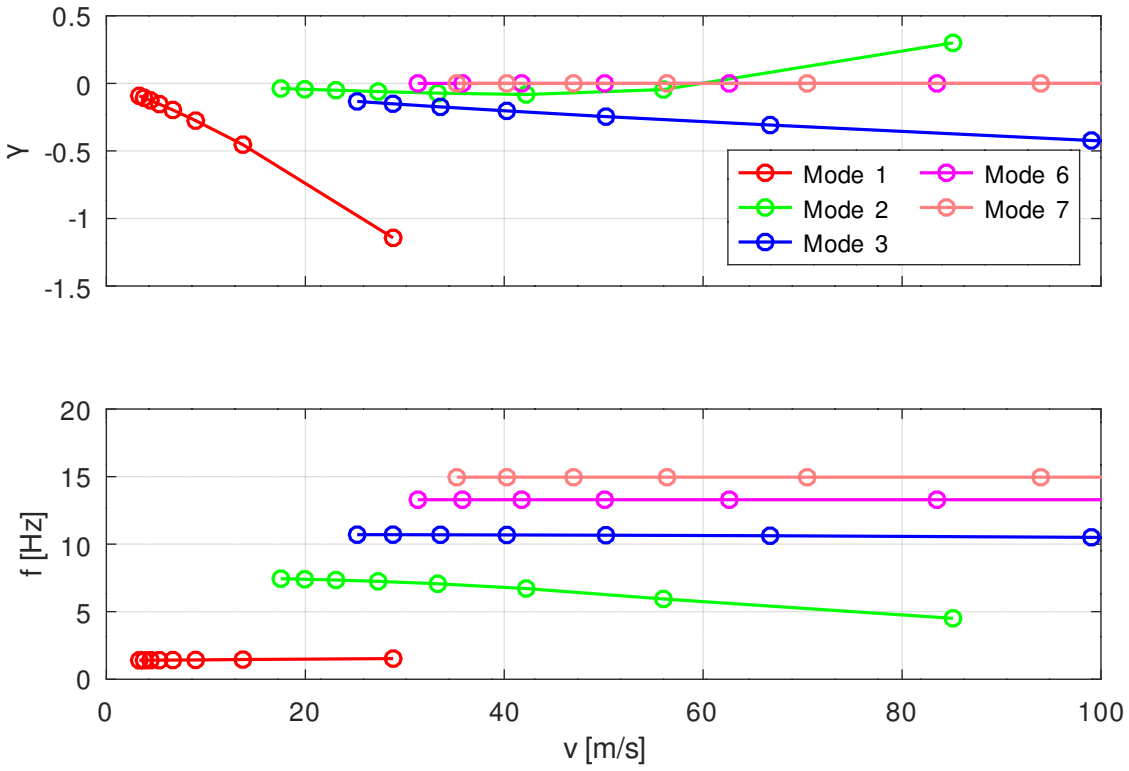


Figure 4.1-7: Flutter Results versus Velocity (k-method)

meter at a reduced frequency of about 1. In Figure 4.1-7 the damping parameter and the frequency are plotted versus the velocity. The zero crossing of mode 2 occurs at a velocity of about 60 m/s.

File `flutter_k2.m` performs a second analysis examining more closely the range of reduced frequencies from 0.5 to 1.0 with an increment of 0.05.

```
# Flutter Example: Flutter analysis using the k-method
#                  Refined set of reduced frequencies
#
# File needed: aeroelastic.bin from aeroelastic.m
#
# ------------------------------------------------------------------

  fid  = fopen("flutter_k2.res", "wt");

# Data

  parameters

  kred = 0.5 : 0.05 : 1.0;   % Reduced frequencies

# Load aeroelastic component

  load aeroelastic.bin

# Flutter analysis

  wing = mfs_flutter(wing, kred, rho, "k");
  save -binary flutter_k2.bin wing

  mfs_print(fid, wing, "flutter", "curves");

  fclose(fid);
```

The results for flutter mode 2 read:

```
Flutter mode  2

Point:      k           v           g           f
-------------------------------------------------------
    1:    0.500000  8.5115e+04   2.995e-01     4.51550
    2:    0.550000  7.9981e+04   2.315e-01     4.66743
    3:    0.600000  7.5737e+04   1.740e-01     4.82159
    4:    0.650000  7.2176e+04   1.251e-01     4.97777
    5:    0.700000  6.9118e+04   8.472e-02     5.13352
    6:    0.750000  6.6404e+04   5.114e-02     5.28428
    7:    0.800000  6.3975e+04   2.303e-02     5.43040
    8:    0.850000  6.1764e+04   3.388e-04     5.57037
    9:    0.900000  5.9708e+04  -1.817e-02     5.70167
   10:    0.950000  5.7795e+04  -3.343e-02     5.82560
   11:    1.000000  5.6000e+04  -4.542e-02     5.94178
```

It can be seen that the zero crossing occurs at a reduced frequency of about 0.85 corresponding to a velocity of about 61.8 m/s. This is also confirmed by Figure 4.1-8 showing the flutter curves created with script `plot_k2.m`.

Finally, to verify the results, a flutter analysis using the pk-method is performed. As this method is more time consuming than the k-method, the velocity range to be examined is restricted to the interval from 60 m/s to 62 m/s, with a velocity increment of 0.1 m/s. In the pk-method, the flutter modes appear as complex conjugate pairs and are sorted according to increasing frequencies. To save time, we restrict iterations to the first five mode pairs. This analysis is performed by GNU Octave script `flutter_pk.m`.

```
# Flutter Example: Flutter analysis using the pk-method
#
# File needed: aeroelastic.bin from aeroelastic.m
#
# ------------------------------------------------------------------

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

# Data

  parameters

  v = 60 : 0.1 : 62;      % Velocities (in m/s)
  pairs = 1 : 5;          % Mode pairs to iterate on

  v = 1000 * v;           % Velocities in mm/s

# Load aeroelastic component
```

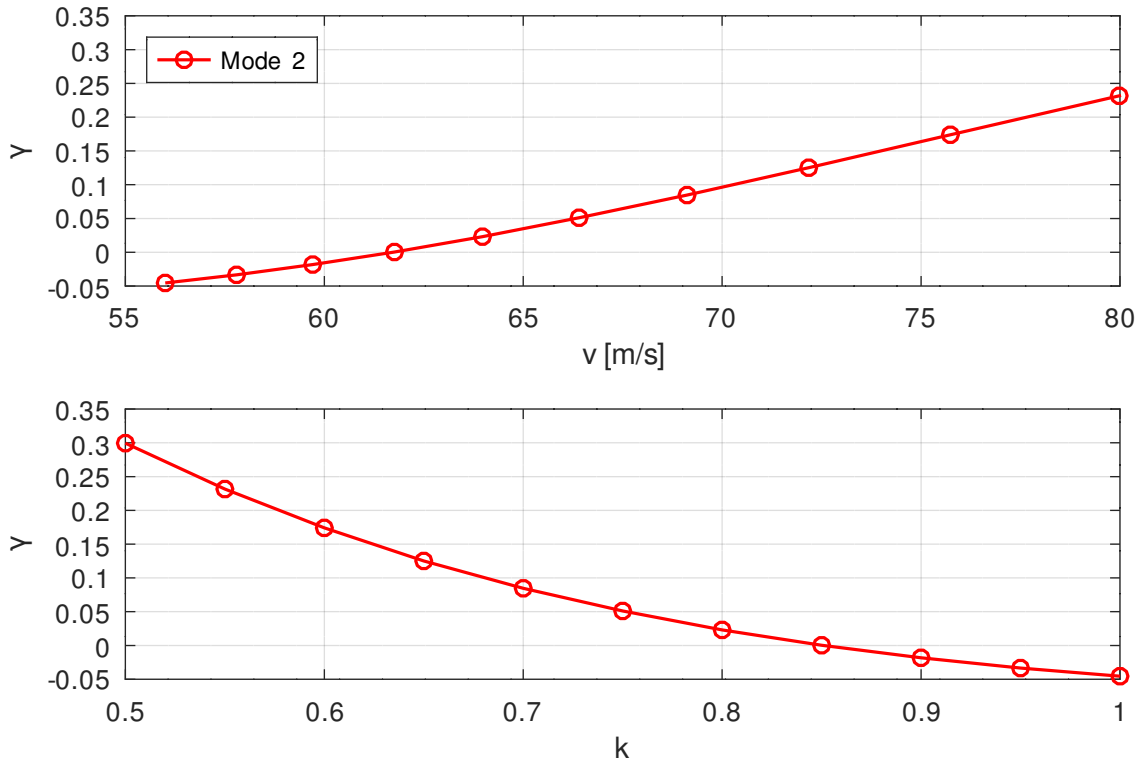


Figure 4.1-8: Damping Parameter of Flutter Mode 2 (k-Method)

```
  load aeroelastic.bin

# Flutter analysis

  wing = mfs_flutter(wing, v, rho, "pk", pairs);
  save -binary flutter_pk.bin wing

  mfs_print(fid, wing, "flutter", "curves");

  fclose(fid);
```

The output file contains the following information:

```
------------------------------------------------------------------------------

Component "wing"

Results of flutter analysis, pk-method

 Flutter mode  1

 Point:      v           k           a           f
 ------------------------------------------------------
     1:  6.0000e+04    0.000000  -3.165e+01     0.00000
     2:  6.0100e+04    0.000000  -3.172e+01     0.00000
     3:  6.0200e+04    0.000000  -3.179e+01     0.00000
     4:  6.0300e+04    0.000000  -3.185e+01     0.00000
     5:  6.0400e+04    0.000000  -3.192e+01     0.00000
     6:  6.0500e+04    0.000000  -3.199e+01     0.00000
     7:  6.0600e+04    0.000000  -3.206e+01     0.00000
     8:  6.0700e+04    0.000000  -3.213e+01     0.00000
     9:  6.0800e+04    0.000000  -3.219e+01     0.00000
    10:  6.0900e+04    0.000000  -3.226e+01     0.00000
    11:  6.1000e+04    0.000000  -3.233e+01     0.00000
    12:  6.1100e+04    0.000000  -3.239e+01     0.00000
    13:  6.1200e+04    0.000000  -3.246e+01     0.00000
    14:  6.1300e+04    0.000000  -3.253e+01     0.00000
    15:  6.1400e+04    0.000000  -3.260e+01     0.00000
    16:  6.1500e+04    0.000000  -3.266e+01     0.00000
    17:  6.1600e+04    0.000000  -3.273e+01     0.00000
    18:  6.1700e+04    0.000000  -3.280e+01     0.00000
    19:  6.1800e+04    0.000000  -3.286e+01     0.00000
    20:  6.1900e+04    0.000000  -3.293e+01     0.00000
    21:  6.2000e+04    0.000000  -3.299e+01     0.00000

 Flutter mode  2

 Point:      v           k           a           f
 ------------------------------------------------------
     1:  6.0000e+04    0.000000  -3.018e+00     0.00000
     2:  6.0100e+04    0.000000  -3.009e+00     0.00000
     3:  6.0200e+04    0.000000  -2.999e+00     0.00000
     4:  6.0300e+04    0.000000  -2.990e+00     0.00000
     5:  6.0400e+04    0.000000  -2.981e+00     0.00000
     6:  6.0500e+04    0.000000  -2.972e+00     0.00000
     7:  6.0600e+04    0.000000  -2.964e+00     0.00000
     8:  6.0700e+04    0.000000  -2.955e+00     0.00000
     9:  6.0800e+04    0.000000  -2.946e+00     0.00000
    10:  6.0900e+04    0.000000  -2.937e+00     0.00000
    11:  6.1000e+04    0.000000  -2.928e+00     0.00000
    12:  6.1100e+04    0.000000  -2.919e+00     0.00000
    13:  6.1200e+04    0.000000  -2.911e+00     0.00000
    14:  6.1300e+04    0.000000  -2.902e+00     0.00000
    15:  6.1400e+04    0.000000  -2.893e+00     0.00000
    16:  6.1500e+04    0.000000  -2.885e+00     0.00000
    17:  6.1600e+04    0.000000  -2.876e+00     0.00000
    18:  6.1700e+04    0.000000  -2.868e+00     0.00000
    19:  6.1800e+04    0.000000  -2.859e+00     0.00000
    20:  6.1900e+04    0.000000  -2.851e+00     0.00000
```

```
   21:  6.2000e+04    0.000000  -2.842e+00     0.00000

Flutter mode  3

Point:      v           k           a           f
-----------------------------------------------------
    1:  6.0000e+04    0.889081  -4.685e-01     5.66006
    2:  6.0100e+04    0.886787  -4.421e-01     5.65487
    3:  6.0200e+04    0.884500  -4.155e-01     5.64967
    4:  6.0300e+04    0.882223  -3.889e-01     5.64449
    5:  6.0400e+04    0.879958  -3.622e-01     5.63933
    6:  6.0500e+04    0.877703  -3.354e-01     5.63419
    7:  6.0600e+04    0.875459  -3.085e-01     5.62908
    8:  6.0700e+04    0.873225  -2.816e-01     5.62398
    9:  6.0800e+04    0.871001  -2.546e-01     5.61890
   10:  6.0900e+04    0.868788  -2.275e-01     5.61384
   11:  6.1000e+04    0.866585  -2.003e-01     5.60880
   12:  6.1100e+04    0.864392  -1.731e-01     5.60378
   13:  6.1200e+04    0.862209  -1.458e-01     5.59877
   14:  6.1300e+04    0.860036  -1.184e-01     5.59379
   15:  6.1400e+04    0.857874  -9.093e-02     5.58883
   16:  6.1500e+04    0.855721  -6.338e-02     5.58388
   17:  6.1600e+04    0.853578  -3.576e-02     5.57895
   18:  6.1700e+04    0.851445  -8.046e-03     5.57405
   19:  6.1800e+04    0.849322   1.975e-02     5.56916
   20:  6.1900e+04    0.847209   4.762e-02     5.56429
   21:  6.2000e+04    0.845106   7.558e-02     5.55945

Flutter mode  4

Point:      v           k           a           f
-----------------------------------------------------
    1:  6.0000e+04   -0.889081  -4.685e-01    -5.66006
    2:  6.0100e+04   -0.886787  -4.421e-01    -5.65487
    3:  6.0200e+04   -0.884500  -4.155e-01    -5.64967
    4:  6.0300e+04   -0.882223  -3.889e-01    -5.64449
    5:  6.0400e+04   -0.879958  -3.622e-01    -5.63933
    6:  6.0500e+04   -0.877703  -3.354e-01    -5.63419
    7:  6.0600e+04   -0.875459  -3.085e-01    -5.62908
    8:  6.0700e+04   -0.873225  -2.816e-01    -5.62398
    9:  6.0800e+04   -0.871001  -2.546e-01    -5.61890
   10:  6.0900e+04   -0.868788  -2.275e-01    -5.61384
   11:  6.1000e+04   -0.866585  -2.003e-01    -5.60880
   12:  6.1100e+04   -0.864392  -1.731e-01    -5.60378
   13:  6.1200e+04   -0.862209  -1.458e-01    -5.59877
   14:  6.1300e+04   -0.860036  -1.184e-01    -5.59379
   15:  6.1400e+04   -0.857874  -9.093e-02    -5.58883
   16:  6.1500e+04   -0.855721  -6.338e-02    -5.58388
   17:  6.1600e+04   -0.853578  -3.576e-02    -5.57895
   18:  6.1700e+04   -0.851445  -8.046e-03    -5.57405
   19:  6.1800e+04   -0.849322   1.975e-02    -5.56916
   20:  6.1900e+04   -0.847209   4.762e-02    -5.56429
   21:  6.2000e+04   -0.845106   7.558e-02    -5.55945

Flutter mode  5

Point:      v           k           a           f
-----------------------------------------------------
    1:  6.0000e+04    1.621397  -1.460e+01    10.32214
    2:  6.0100e+04    1.618525  -1.462e+01    10.32102
    3:  6.0200e+04    1.615662  -1.465e+01    10.31991
    4:  6.0300e+04    1.612808  -1.467e+01    10.31879
    5:  6.0400e+04    1.609964  -1.470e+01    10.31768
    6:  6.0500e+04    1.607129  -1.472e+01    10.31656
    7:  6.0600e+04    1.604303  -1.474e+01    10.31544
    8:  6.0700e+04    1.601486  -1.477e+01    10.31432
    9:  6.0800e+04    1.598679  -1.479e+01    10.31321
   10:  6.0900e+04    1.595881  -1.482e+01    10.31209
   11:  6.1000e+04    1.593092  -1.484e+01    10.31097
   12:  6.1100e+04    1.590312  -1.487e+01    10.30985
   13:  6.1200e+04    1.587541  -1.489e+01    10.30873
   14:  6.1300e+04    1.584779  -1.492e+01    10.30761
   15:  6.1400e+04    1.582026  -1.494e+01    10.30649
   16:  6.1500e+04    1.579282  -1.497e+01    10.30537
```

```
   17:  6.1600e+04    1.576546  -1.499e+01    10.30425
   18:  6.1700e+04    1.573820  -1.502e+01    10.30313
   19:  6.1800e+04    1.571102  -1.504e+01    10.30200
   20:  6.1900e+04    1.568393  -1.507e+01    10.30088
   21:  6.2000e+04    1.565692  -1.509e+01    10.29976

Flutter mode  6

Point:      v           k           a           f
-------------------------------------------------------
    1:  6.0000e+04   -1.621397  -1.460e+01   -10.32214
    2:  6.0100e+04   -1.618525  -1.462e+01   -10.32102
    3:  6.0200e+04   -1.615662  -1.465e+01   -10.31991
    4:  6.0300e+04   -1.612808  -1.467e+01   -10.31879
    5:  6.0400e+04   -1.609964  -1.470e+01   -10.31768
    6:  6.0500e+04   -1.607129  -1.472e+01   -10.31656
    7:  6.0600e+04   -1.604303  -1.474e+01   -10.31544
    8:  6.0700e+04   -1.601486  -1.477e+01   -10.31432
    9:  6.0800e+04   -1.598679  -1.479e+01   -10.31321
   10:  6.0900e+04   -1.595881  -1.482e+01   -10.31209
   11:  6.1000e+04   -1.593092  -1.484e+01   -10.31097
   12:  6.1100e+04   -1.590312  -1.487e+01   -10.30985
   13:  6.1200e+04   -1.587541  -1.489e+01   -10.30873
   14:  6.1300e+04   -1.584779  -1.492e+01   -10.30761
   15:  6.1400e+04   -1.582026  -1.494e+01   -10.30649
   16:  6.1500e+04   -1.579282  -1.497e+01   -10.30537
   17:  6.1600e+04   -1.576546  -1.499e+01   -10.30425
   18:  6.1700e+04   -1.573820  -1.502e+01   -10.30313
   19:  6.1800e+04   -1.571102  -1.504e+01   -10.30200
   20:  6.1900e+04   -1.568393  -1.507e+01   -10.30088
   21:  6.2000e+04   -1.565692  -1.509e+01   -10.29976

            …           …          …           …
```

The first two flutter modes are real modes with a negative decay rate (column **a**). Hence, they are stable. A positive decay rate, together with a zero frequency, would indicate a static divergence.

The remaining flutter modes are complex conjugate pairs, with a positive and a negative frequency. It can be seen that flutter mode 3 becomes unstable at a velocity of about $61.7\ \mathrm{m/s}$, confirming the results obtained with the k-method.

GNU Octave script `post_pk.m` plots the flutter parameters of flutter mode 3 and extracts the flutter mode shape of flutter mode 3 at flutter point 18 which is closest to the zero crossing of the decay rate. Both the flutter mode shape of the solid and the aerodynamic mesh are exported to Gmsh and subsequently merged into one file `wing_pk.dsp` that can be visualized together with file `wing.msh`.

```
# Flutter Example: Postprocessing of results from pk-method
#
# - plot flutter curves
# - compute and export flutter shape
#
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

  modpnt = [3, 18];   % Flutter mode, flutter point
```

```
  ltext = sprintf("Mode %d", modpnt(1));

  load flutter_pk.bin

# Flutter curves

  [v, a, kred, f] = mfs_getresp(wing, "flutter",
                                "curves", modpnt(1));
  v = v / 1000;

  figure(1, "position", [50, 50, 1000, 500],
            "paperposition", [0, 0, 17, 11]);
  subplot(2, 1, 1)
     plot(v, a, "color", "red", "marker", "o");
     legend(ltext, "location", "northwest");
     grid;
     axis("labely");
     ylabel("a [1/s]");
  subplot(2, 1, 2)
     plot(v, f, "color", "red", "marker", "o");
     grid;
     xlabel("v [m/s]"); ylabel("f [Hz]");
  print(["flutter_pk", EXT], FORMAT);

# Back tranformation

  [wings, winga] = mfs_back(wing, "flutter", "disp", modpnt);
```

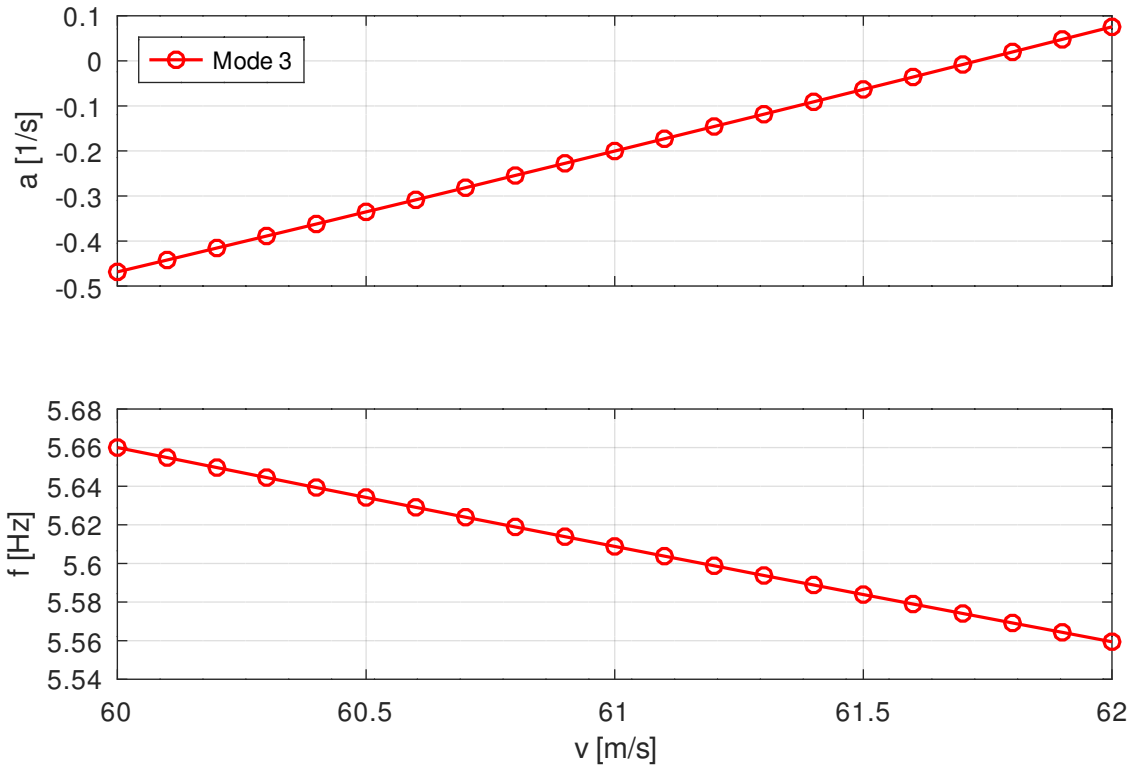


Figure 4.1-9: Flutter Parameters of Mode 3 (pk-Method)

```
# Export

  mfs_export("solid_pk.dsp", "msh", wings, "flutter", "disp");
  mfs_export("aero_pk.dsp", "msh", winga, "flutter", "disp");

# Merge

  mfs_merge("solid_pk.dsp", "aero_pk.dsp", "wing_pk.dsp", "msh");
```

The plot generated by this code can be seen in Figure 4.1-9.

Post-processing

To animate the flutter mode shape in Gmsh, we proceed as follows:

1. We start Gmsh, open file `wing.msh` and merge file `wing_pk.dsp`. View 0 contains both the real and the imaginary part of the flutter mode shape. We can use the right and left arrow keys to toggle between the real and the imaginary part. After adjusting some of the options, we get the picture shown in Figure 4.1-10. The rotation angles have been set to 300, 0, 320 using the manipulator menu (`Tools → Manipulator`).
2. Next, we 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 4.1-11).

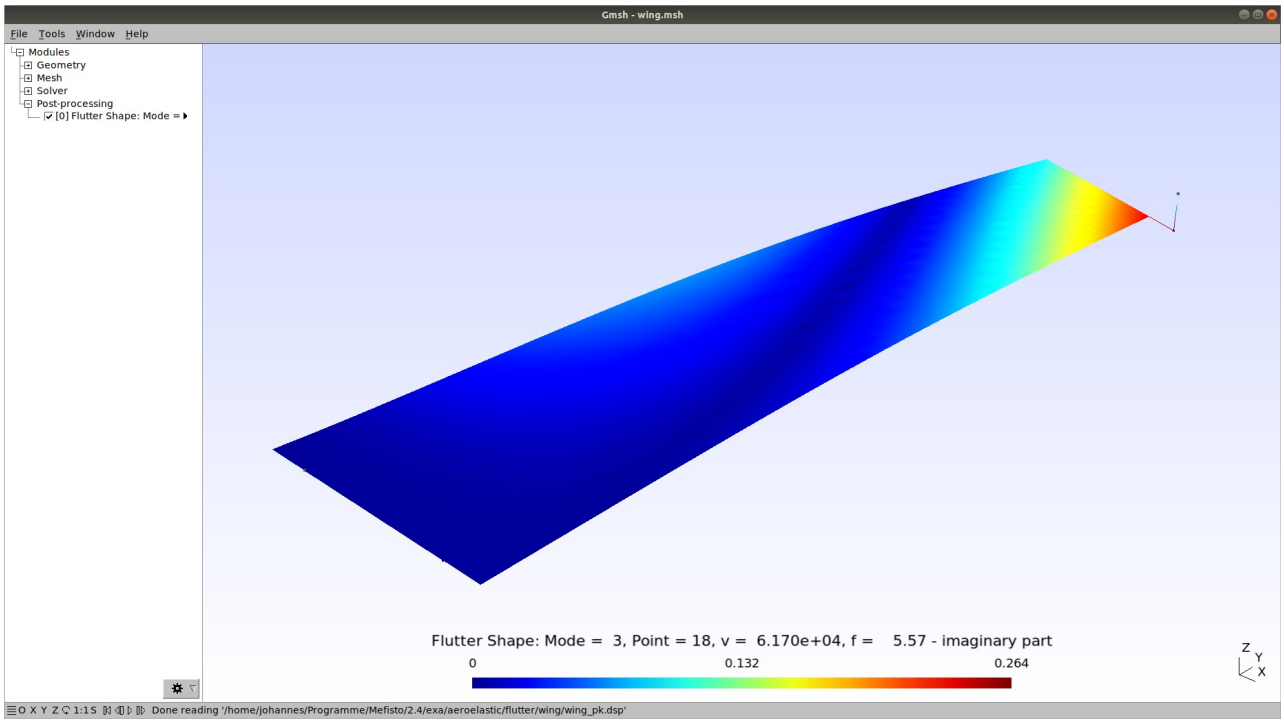


Figure 4.1-10: Complex Flutter Mode Shape

3. The plugin creates a new view with number 1 containing all time steps. We can use the right and left arrow keys to go through the time steps. Figure 4.1-12 shows the deformation at time step 13.
4. Finally, we can use the animation keys at the bottom left of the screen

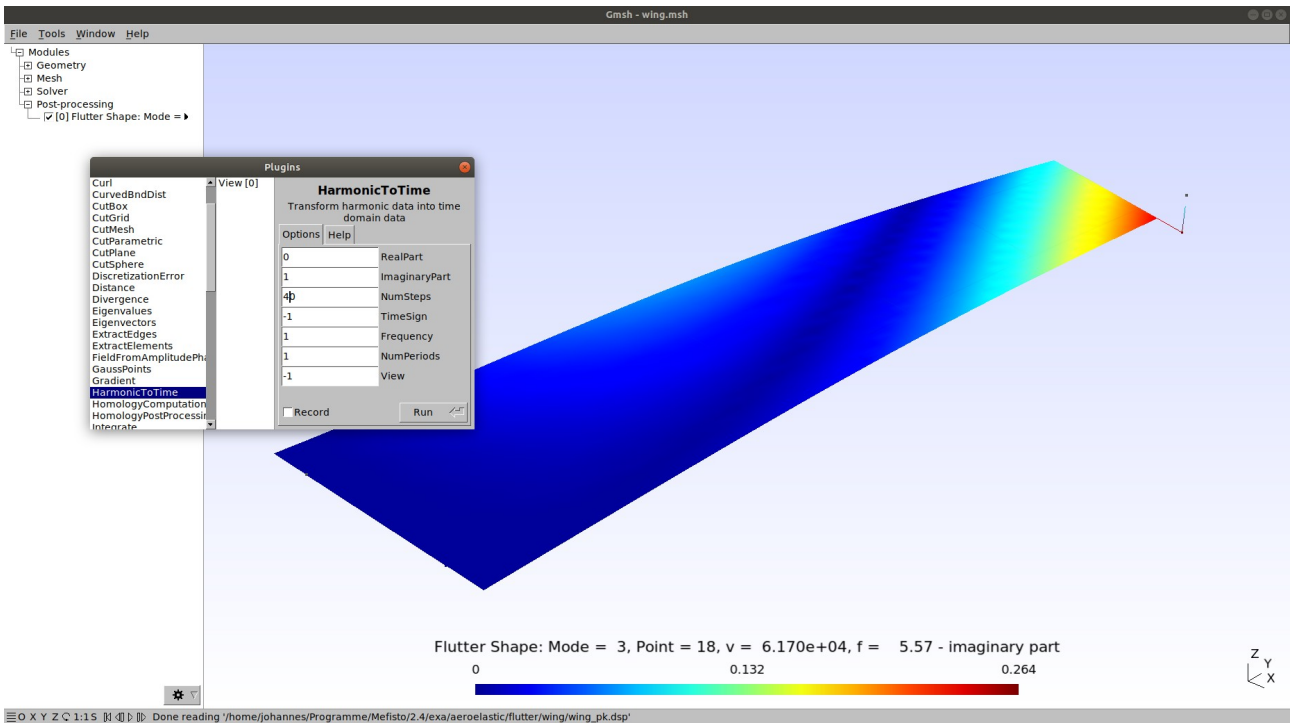


Figure 4.1-11: HarmonicToTime Plugin

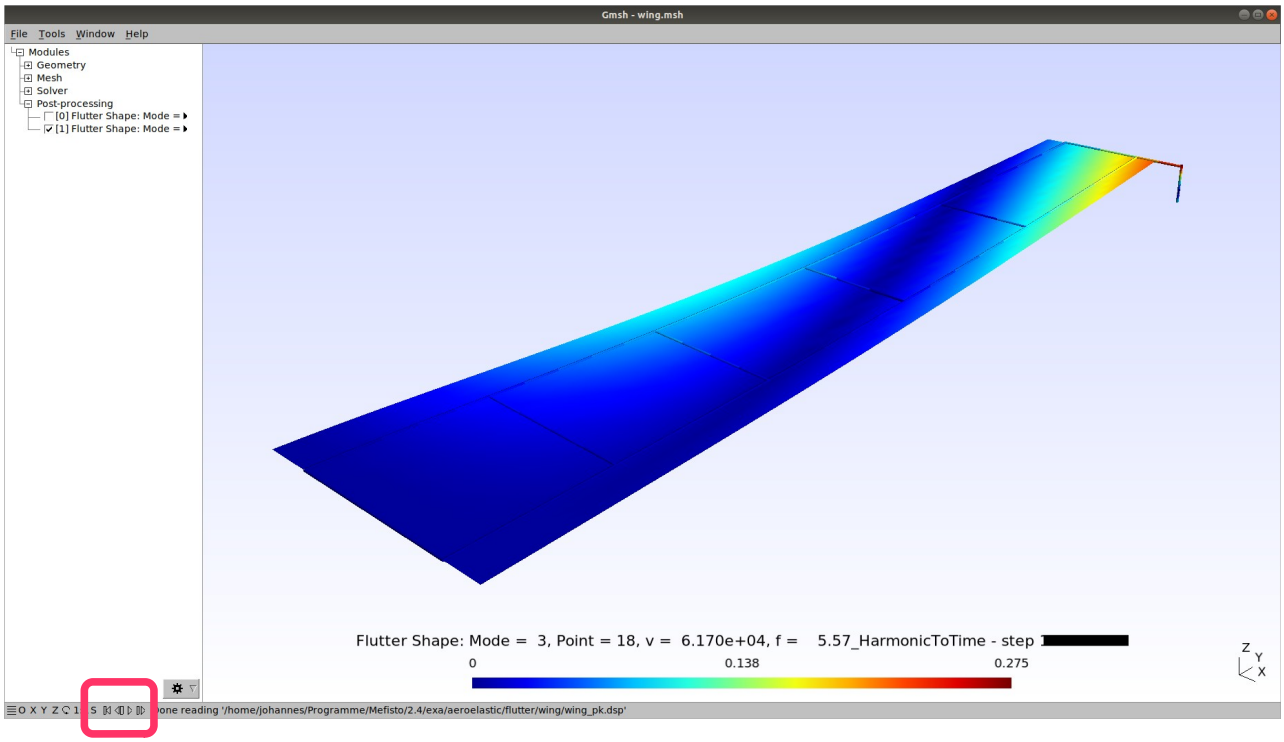


Figure 4.1-12: Deformation at Time Step 13

(see Figure 4.1-12) to start and stop an animation.

5 Frequency Response Analysis

5.1 Glider

Summary

Directory:	`exa/aeroelastic/freqresp/glider`
Objectives:	• learn how to determine the cut-off frequency of a time-dependent excitation • learn how to compute the response of an aircraft to a gust • learn how to compute the response of an aircraft to a control surface motion • learn how to compute transient results from frequency response results • learn how to animate the transient results
Elements:	**`b2, m1`**
Method:	Vortex-Lattice
Functions:	**`mfs_line, mfs_linenodes, mfs_beamsection, mfs_new, mfs_stiff, mfs_mass, mfs_freevib, mfs_massproperties, mfs_export, mfs_print, mfs_splines, mfs_transfer, mfs_freqresp, mfs_merge, mfs_extract, mfs_getresp, mfs_freq2time, mfs_back`**

Problem Description

Compute the response of a standard class glider to two different gust loads and to an elevator input. The standard class glider is the same as that used in Example 3.1, see Figure 3.1-1. For damping, use modal damping with a damping ratio of 2 % for all normal modes. The mass density of the air is $1.21\ \mathrm{kg/m^3}$.

First, examine the response to two different gusts. The profile of the first gust reads

$$w_1^G(x_E) = \frac{w_0}{2}\left(1-\cos\left(2\pi\frac{x_E}{L}\right)\right) = w_0 \sin^2\left(\pi\frac{x_E}{L}\right), \quad 0 \le x_E \le L \quad .$$

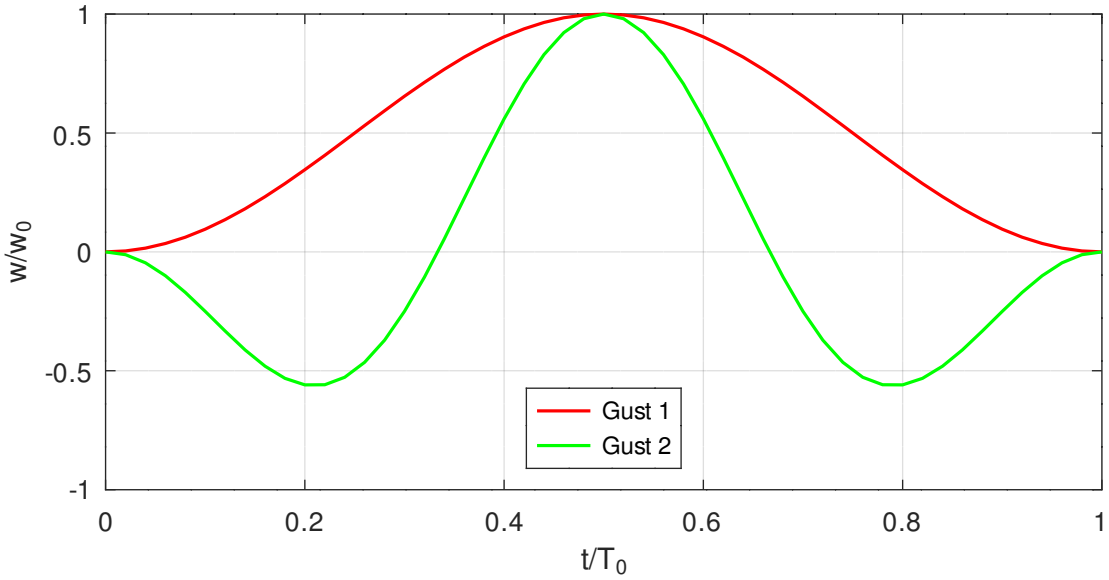


Figure 5.1-1: Gust Profiles

This gust profile is used as standard for the design of aircrafts. The second gust profile is

$$w_2^G(x_E) = w_0\left(\sin^2\left(\pi\frac{x_E}{L}\right) - \sin^2\left(2\pi\frac{x_E}{L}\right)\right), \quad 0 \le x_E \le L$$

which is a more realistic profile because in practice, downdrafts precede and follow updrafts. In these equations, w_0 is the gust amplitude, x_E is the distance travelled in a coordinate system fixed to the ground, and L is the gust length. For the analysis, use a gust amplitude of $1\ \mathrm{m/s}$ and a gust length of $30\ \mathrm{m}$. Both gust profiles can be seen in Figure 5.1-1 where

$$T_0 = L/v$$

is the time it takes to fly through the gust with flight velocity v. Because of

$$x_E = v\,t$$

we have

$$t/T_0 = x_E/L \quad .$$

For flight velocities of $30\ \mathrm{m/s}$ and $60\ \mathrm{m/s}$, compute

- the absolute and relative vertical displacements at the wing root (WR), the wing tip (WT) and the tail,
- the vertical accelerations at the wing root, the wing tip and the tail,
- the wing root bending moment.

For the second gust and a flight velocity of $60\ \mathrm{m/s}$, perform a back transformation and export the displacements to Gmsh so that you can create an anima-

tion.

Next, examine the response to the elevator input

$$\eta_E(t) = \eta_{E0} \sin^3\left(2\pi \frac{t}{T_0}\right), \quad 0 \le t \le T_0$$

with $\eta_{E0} = 5°$ and $T_0 = 1\ \mathrm{s}$, see Figure 5.1-4. The exponent 3 of the sine function causes the speed of the elevator deflection to be zero at the beginning and end.

For a flight velocity of $40\ \mathrm{m/s}$, compute the vertical displacements and accelerations at the wing root, the wing tip and the tail as well as the wing root bending moment.

Analysis of the Loads

First, the cut-off frequency of the time-dependent loads must be determined. Both the mesh size and the number of normal modes to be used in modal reduction depend on the cut-off frequency.

The cut-off frequency can be determined from the Fourier transform. Contributions from frequencies above the cut-off frequency are considered negligible.

The GNU Octave scripts to determine the cut-off frequency are located in directory `loads`. GNU Octave script `gust.m` computes and plots the Fourier transforms of the gusts. It starts with the definition of the gust parameters and the time step.

```
# Example: Frequency response analysis of a standard class glider
#          Fourier transform of the gusts
#          Fourier transforms are stored in file gust.bin.
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

# Data of the gusts (m, s)

  L  = 30;         % Gust length
  w0 = 1;          % Gust velocity amplitude
  v  = [30, 60];   % Flight velocities
  dt = 0.01;       % Time step
```

Next, some important parameters are computed, and the arrays for the Fourier transforms are initialised. Index 1 denotes the first gust and index 2 the second.

First, the duration of the period to be considered is defined. This duration depends on two criteria. First, it must be long enough for the response to go to zero. We assume that a value of five times the time it takes the glider to pass the gust is sufficient. To keep things simple, the same value is used for both velocities. Second, the frequency increment is inversely proportional to the duration of the period considered. If we want to achieve a finer frequency resolution, we need to increase the duration.

Next, the Nyquist frequency and the frequency increment are defined. The Nyquist frequency is inversely proportional to the time step. The Fourier transform will contain frequencies up to the Nyquist frequency.

```
# Initialization

  T    = 5 * L / v(1);  % Duration
  N    = T / dt;        % Length of time series
  fmax = 1 / (2 * dt);  % Nyquist frequency
  df   = 1 / T;         % Frequency increment
  f    = 0 : df : fmax; % Frequencies
  nf   = length(f);     % Number of frequencies

  W1 = zeros(2, N); W2 = zeros(2, N);
```

The Fourier transforms are computed in a loop over the two flight velocities. First, the time to pass the gust is determined.

```
# Loop over velocities and compute Fourier transforms

  for n = 1 : 2

      text{n} = sprintf("v = %2.0f m/s", v(n));
      T0      = L / v(n); % Time to pass the gust
```

Next, the time series corresponding to the two gusts are computed. These computations need be done only for the time to pass the gust, the remaining values being zero.

```
#   Time series

      t0 = 0 : (dt / T0) : 1;
      w1 = w0 * sin(pi * t0).^2;
      w2 = w1 - w0 * sin(2 * pi * t0).^2;
```

Now, the Fourier transforms can be computed. GNU Octave function **`fft`** returns the coefficients of the discrete Fourier transform. These values have to be multiplied with the time step to obtain a discrete approximation of the continuous Fourier transform. The additional argument **`N`** requests the computed time series to be zero padded up to a length of N.

```
#   Fourier transforms

      W1(n, :) = dt * fft(w1, N);
      W2(n, :) = dt * fft(w2, N);

  endfor
```

Subsequently, both the original time series and the Fourier transforms are plotted. Please note that only the first **nf** entries of arrays **W1** and **W2** are plotted. These values correspond to positive frequencies. The remaining fields contain the complex conjugate values corresponding to negative frequencies.

```
# Plot data and results

  figure(1, "position", [100, 100, 800, 400],
            "paperposition", [0, 0, 17, 8]);
  axes("outerposition", [0, 0.01, 1, 1]);
  plot(t0, w1,t0, w2);
  legend("Gust 1", "Gust 2", "location", "south");
  grid;
  xlabel("t/T_0"); ylabel("w/w_0");
  print(["gust", EXT], FORMAT);

  figure(2, "position", [100, 100, 800, 400],
            "paperposition", [0, 0, 17, 10]);
  subplot(2, 1, 1);
     plot(f, abs(W1(1, 1 : nf)), f, abs(W2(1, 1 : nf)));
     legend("Gust 1", "Gust 2");
     title(text{1});
     grid;
     xlim([0, 10]);
     ylabel("|W| [m/s/Hz]");
  subplot(2, 1, 2);
     plot(f, abs(W1(2, 1 : nf)), f, abs(W2(2, 1 : nf)));
     title(text{2});
     grid;
     xlim([0, 10]);
     ylabel("|W| [m/s/Hz]");
     xlabel("f [Hz]");
  print(["gust_ft", EXT], FORMAT);
```

Finally, the Fourier transforms together with the frequency increment and the velocities are stored in file `gust.bin`. These data will be needed to compute the Fourier transforms of the responses.

```
# Save Fourier transforms for later use

  W{1} = W1; W{2} = W2;
  save -binary gust.bin W df v
```

The resulting Fourier transforms can be seen in Figure 5.1-2. The figure shows that the Fourier transforms are practically zero for frequencies above 10 Hz, indicating that a cut-off frequency of 10 Hz should be appropriate.

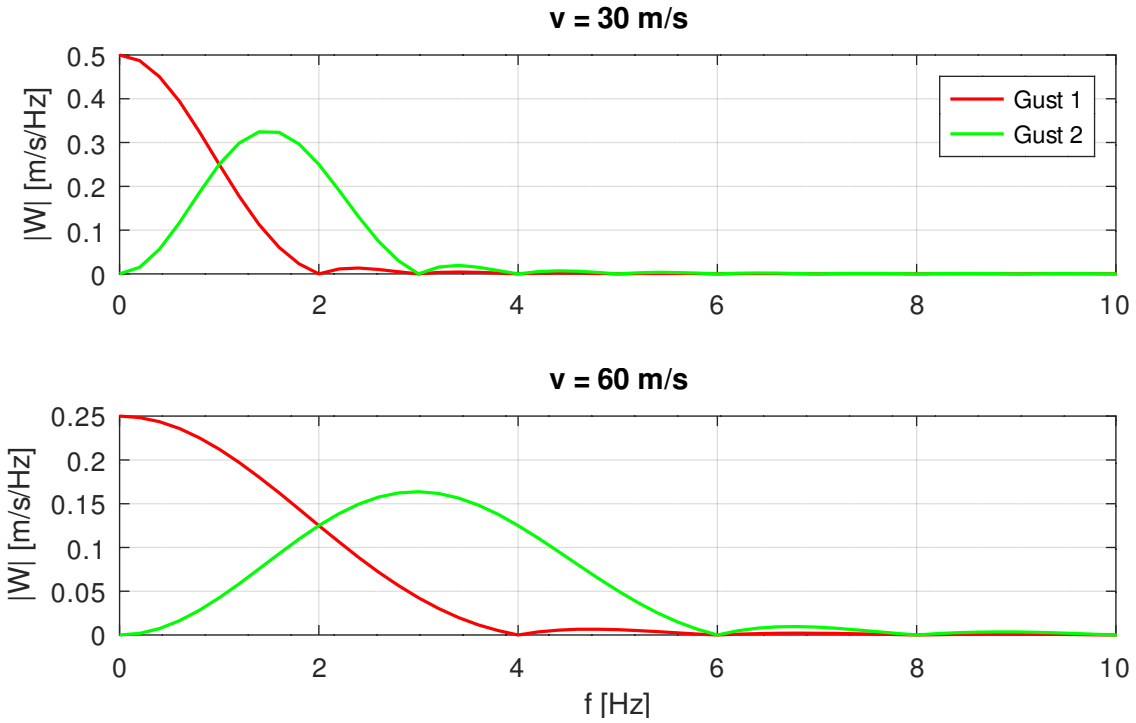


Figure 5.1-2: Fourier Transforms of the Gusts

We can use the GNU Octave function **`resample`**, contained in the `signal` package, to study the effect of truncating the Fourier transform at 10 Hz. With this function, we can decrease (downsample) or increase (upsample) the sampling rate. If the sampling rate is decreased, the function automatically uses a low-pass filter to make sure the Nyquist condition is satisfied. It is sufficient to perform this analysis for the second gust and a velocity of 60 m/s only.

File `gust_check.m` first loads the `signal` package. Subsequently, the cut-off frequency and the gust parameters are defined, and the Nyquist frequency and the time to pass the gust are determined.

```
# Example: Frequency response analysis of a standard class glider
#          Check cut-off frequency of gust
# ------------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

  pkg load signal

# Cut-off frequency

  fc = 10;

# Data of the gust (m, s)
```

```
  L  = 30;         % Gust length
  w0 = 1;          % Gust velocity amplitude
  v  = 60;         % Flight velocity
  dt = 0.01;       % Time step
  P  = 5;          % Upsampling factor

# Initialization

  fmax = 1 / (2 * dt);  % Nyquist frequency
  T0   = L / v;         % Time to pass the gust
```

Next, the original time series is computed.

```
# Original time series

  t = 0 : (dt / T0) : 1;
  w = w0 * (sin(pi * t).^2 - sin(2 * pi * t).^2);
```

Now, the time series is downsampled so that the new Nyquist frequency matches the cut-off frequency. Subsequently, the downsampled time series is upsampled by increasing the sampling rate by a factor of 5 to get a smoother curve. Upsampling uses the Whittaker-Shannon interpolation formula to interpolate between the given values of the time series.

```
# Filtered time series

  wc = resample(w, fc, fmax);
  fs = 2 * fc; fsT0 = fs * T0;            % Sampling rate
  tc = (0 : length(wc) - 1) / fsT0;       % t/T0
  wp = resample(wc, P, 1);
  tp = (0 : length(wp) - 1) / (P * fsT0); % t/T0
```

Finally, all three time series are plotted in one diagram.

```
# Plot results

  figure(1, "position", [100, 100, 800, 400],
            "paperposition", [0, 0, 17, 10]);
  axes("outerposition", [0, 0.01, 1, 1]);
  plot(t, w, tc, wc, tp, wp);
  legend("Original", "Filtered", "Upsampled",
         "location", "south");
  grid; xlim([0, 1]);
  xlabel("t/T_0"); ylabel("w [m/s]");
  print(["gust_check", EXT], FORMAT);
```

Figure 5.1-3 shows that the values of the downsampled time series (green curve) agree well with those of the original time series. The upsampled curve (blue curve) matches the original curve (red curve), confirming that a cut-off frequency of $10\ \mathrm{Hz}$ is appropriate.

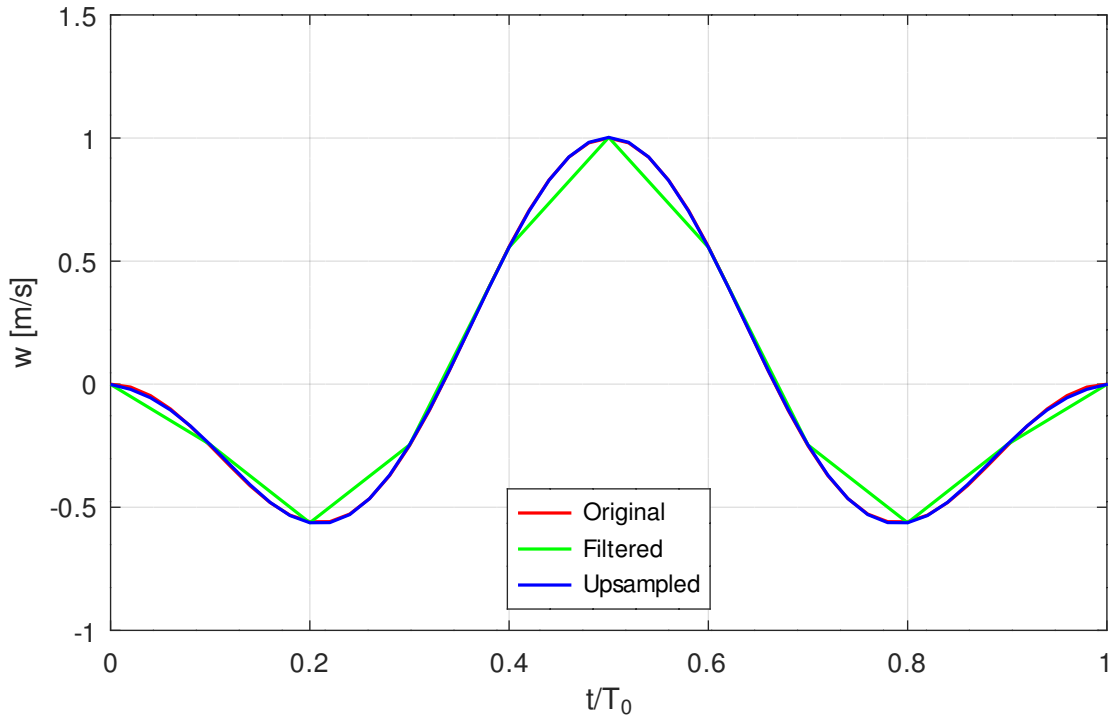


Figure 5.1-3: Test of Gust Cut-off Frequency

The same studies are also performed for the elevator input. GNU Octave script `elevator.m` computes the Fourier transform of the elevator input and plots both the elevator input and its Fourier transform.

```
# Example: Frequency response analysis of a standard class glider
#          Fourier transform of the elevator input
#          Fourier transforms are stored in file elevator.bin.
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

# Data of elevator input (degrees, s)

  eta0 =      5;     % Amplitude of elevator angle
  T0   =    1.0;     % Duration of elevator motion
  dt   =   0.01;     % Time step
  T    = 5 * T0;     % Total duration

# Time series

  N  = T / dt;     % Length of time series

  t0  = 0 : (dt / T0) : 1;
  eta = eta0 * sin(2 * pi * t0).^3;

# Fourier transform
```

```
  ETA = dt * fft(eta, N);

  fmax = 1 / (2 * dt); df   = 1 / T;
  f    = 0 : df : fmax;
  nf   = length(f);

# Plot

  figure(1, "position", [400, 200, 800, 400],
            "paperposition", [0, 0, 17, 12]);
  subplot(2, 1, 1);
     plot(t0, eta, "color", "green");
     grid;
     xlabel("t/T_0"); ylabel('\eta_E [\deg]');
  subplot(2, 1, 2);
     plot(f, abs(ETA(1 : nf)), "color", "red");
     grid; xlim([0, 10]);
     xlabel("f [Hz]"); ylabel('|FT(\eta_E)| [\deg/Hz]');
  print(["elevator", EXT], FORMAT);

  save -binary elevator.bin df ETA
```

The resulting plots can be seen in Figure 5.1-4. The Fourier transform shows that the cut-off frequency of $10\ \mathrm{Hz}$ is also appropriate for the elevator input.

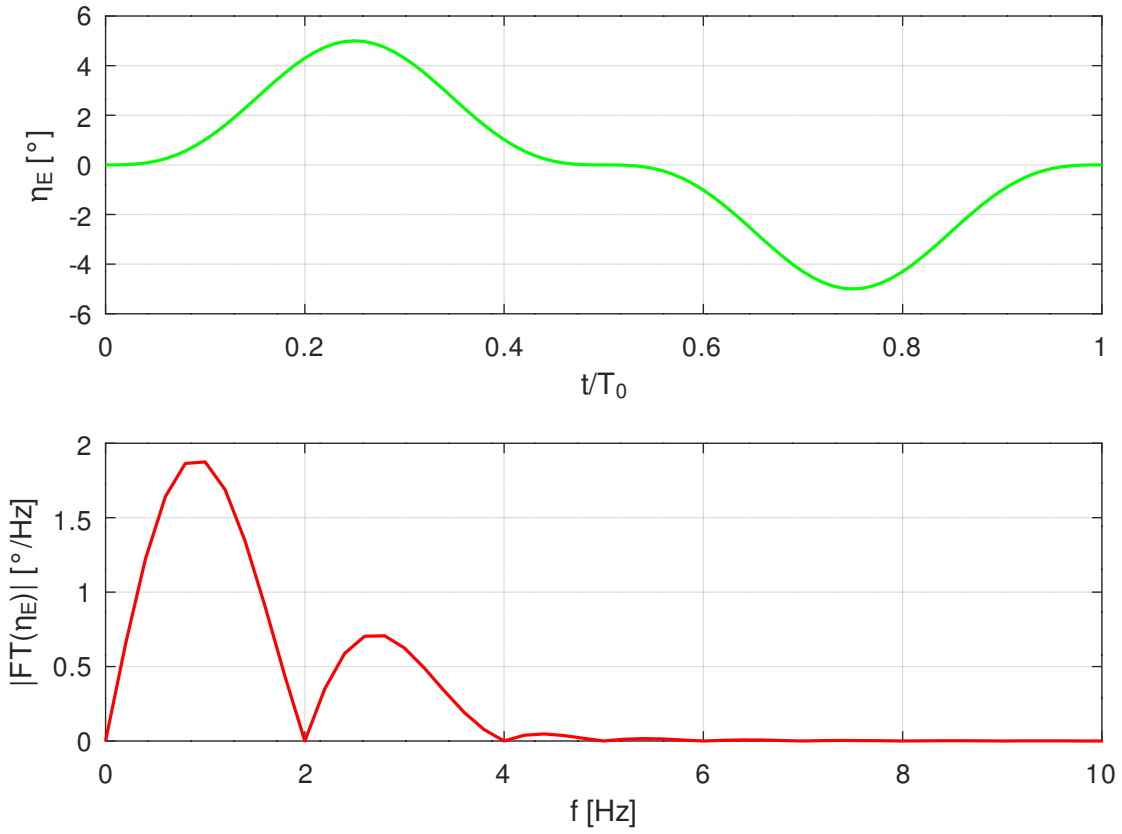


Figure 5.1-4: Elevator Input and its Fourier Transform

Finally, GNU Octave script `elevator_check.m` studies the effect of truncating the Fourier transform at 10 Hz.

```
# Example: Frequency response analysis of a standard class glider
#          Check cut-off frequency of elevator input
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

  pkg load signal

# Cut-off frequency

  fc = 10;

# Data of elevator input (degrees, s)

  eta0 =    5;    % Amplitude of elevator angle
  T0   =  1.0;    % Duration of elevator motion
  dt   = 0.01;    % Time step
  P    =    2;    % Upsampling factor

# Initialization

  fmax = 1 / (2 * dt);  % Nyquist frequency

# Original time series

  t   = 0 : (dt / T0) : 1;
  eta = eta0 * sin(2 * pi * t).^3;

# Filtered time series

  etac = resample(eta, fc, fmax);
  fs   = 2 * fc; fsT0 = fs * T0;
  tc   = (0 : length(etac) - 1) / fs;
  etap = resample(etac, P, 1);
  tp   = (0 : length(etap) - 1) / (P * fs);

# Plot results

  figure(1, "position", [100, 100, 800, 400],
            "paperposition", [0, 0, 17, 7]);
  axes("outerposition", [0, 0.01, 1, 1]);
  plot(t,  eta, tc, etac, tp, etap);
  legend("Original", "Filtered", "Upsampled",
         "location", "northeast");
  grid;
  xlim([0, 1]);
  xlabel("t/T_0"); ylabel('\eta_E [\deg]');
  print(["elevator_check", EXT], FORMAT);
```

Figure 5.1-5 shows a very good agreement between the original time series

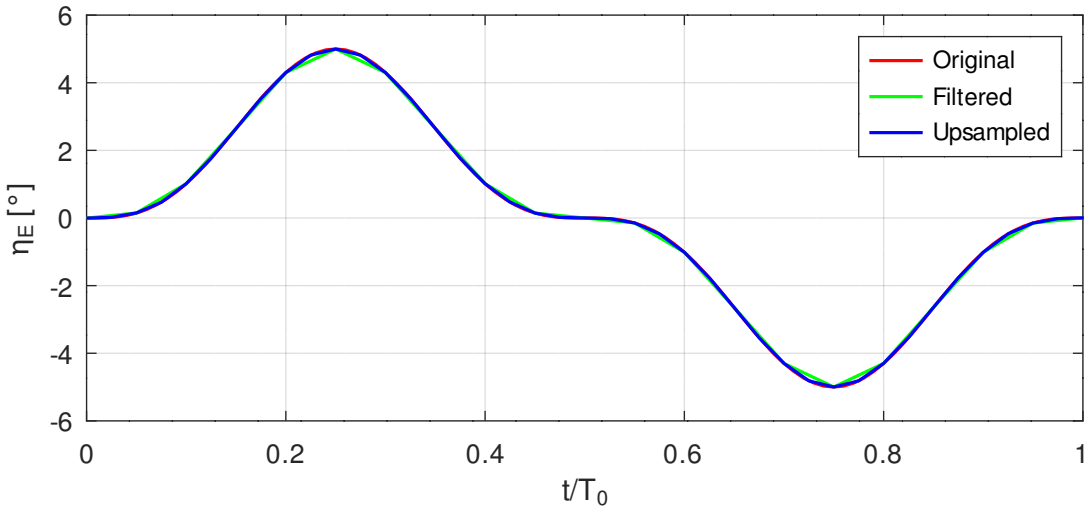


Figure 5.1-5: Test of Elevator Cut-off Frequency

and the time series obtained from the truncated Fourier transform.

Model Definition

The files defining the models are located in directory `models`. This directory also contains the GNU Octave scripts to compute the normal modes and to check the aerodynamic model and the splines.

The common geometry data of the solid and the aerodynamic model are defined in file `geodata.m`. This is the same file as used in Example 3.1.

The **solid model** is defined by function **`solid`** contained in file `solid.m`. The function returns a structure with the model definition and a structure with the node sets.

The definition of the model is largely the same as in Example 3.1. However, we have to remove the definition of the constraints and to add the definition of the damping. The following command defines a constant damping ratio of 2 % for all normal modes:

```
# Damping
# -------

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

GNU Octave script `modes.m` computes the first 15 normal modes and stores the results together with the node sets in file `solid.bin`.

```
# Example: Frequency response analysis of a standard class glider
#          Normal modes analysis
#          Results are stored in file solid.bin.
# ------------------------------------------------------------
```

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

# Number of normal modes

  nmodes = 15;

# Model definition

  model = solid();

# Create and export component

  gliders = mfs_new(fid, model);
  mfs_export("solid.msh", "msh", gliders, "mesh");

# Matrices

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

# Normal modes

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

# Save data

  save -binary solid.bin gliders

  fclose(fid);
```

Of course, the first six normal modes are rigid body modes. Table 5.1-1 lists the frequencies and the shapes of the elastic modes. The frequencies of the

No.	Frequency	Shape
7	4,13 Hz	1st vertical wing bending mode
8	5,68 Hz	1st horizontal fuselage bending mode
9	7,34 Hz	1st horizontal wing bending mode
10	12,31 Hz	1st vertical fuselage bending mode
11	13,50 Hz	1st fuselage torsion mode
12	15,99 Hz	2nd vertical wing bending mode
13	22,45 Hz	2nd horizontal fuselage bending mode
14	22,94 Hz	3rd vertical wing bending mode
15	28,46 Hz	3rd horizontal fuselage bending mode

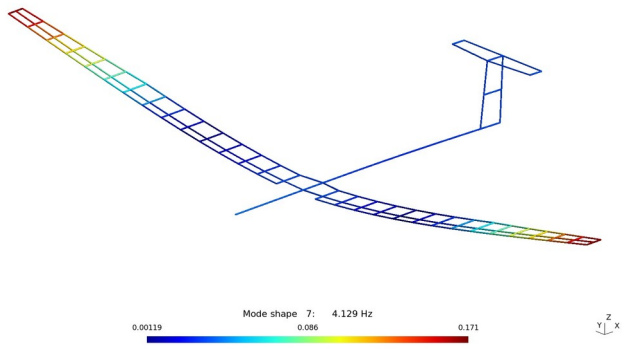
Table 5.1-1: Elastic Modes of the Glider

neglected modes are larger than about three times the cut-off frequency. Thus, their response is quasi-static. Hence, the 15 normal modes computed are sufficient for the modal reduction. Because all excitations are symmetric with respect to the *xz*-plane, only the symmetric modes will be excited. These modes can be seen in Figure 5.1-6. The figure also shows that the mesh is fine enough to accurately represent all calculated modes.

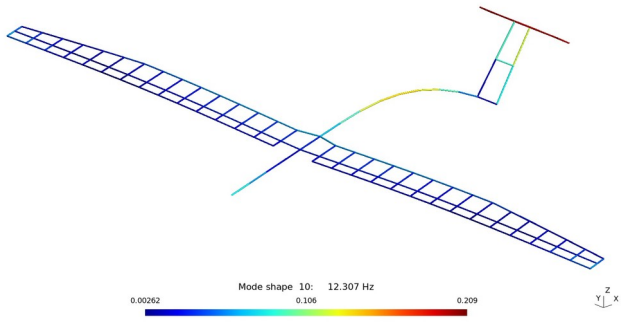
Function **`aero`** contained in file `aero.m` returns the structure defining the **aerodynamic model**. The definition is the same as in Example 3.1. However, the definition of the configurations has been removed because it is not needed. As in an unsteady analysis all results are relative to a static equilibrium, neither the camber nor the rigging angle of incidence need be defined. Hence, these definitions have been removed, too. Actually, it is not necessary to remove these definitions which are only needed in a steady analysis. If present, they are just ignored.

In an unsteady analysis, a reference chord length has to be defined. The reference chord length is used to compute the reduced frequency. If the chord length at the wing root is used as the reference chord length, the model type, subtype and reference chord length are defined by the following command:

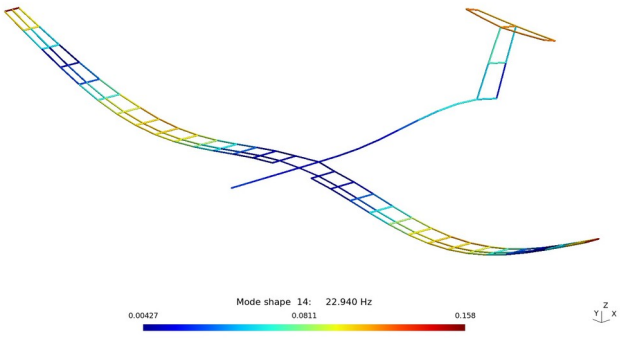
```
model = struct("type", "aero", "subtype", "vlm", "cref", cr);
```



Mode 7: 4.13 Hz



Mode 10: 12.31 Hz



Mode 14: 22.94 Hz

Figure 5.1-6: Symmetric Elastic Modes of the Glider

Finally, function **splinedef** contained in file splinedef.m returns the structure defining the splines. The definition of the splines is exactly the same as in Example 3.1.

GNU Octave script checks.m checks the aerodynamic model and the splines. First, the aerodynamic component is created, exported to Gmsh and saved in binary file aero.bin.

```
# Example: Frequency response analysis of a standard class glider
#          Check the aerodynamic mesh and the splines
#          File needed:   solid.bin
#          Files created: aero.bin
# ------------------------------------------------------------

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

# Build aerodynamic component

  modela  = aero();
  glidera = mfs_new(fid, modela);
  mfs_export("aero.msh", "msh", glidera, "mesh");

  save -binary aero.bin glidera
```

Next, the aeroelastic model is defined, the aeroelastic component is created and the splines are computed.

```
# Define aeroelastic model

  load solid.bin

  model = struct("type", "aeroelastic",
                 "solid", gliders, "aero", glidera,
                 "splines", splinedef());

# Build aeroelastic component and compute splines

  glider = mfs_new(fid, model);
  glider = mfs_splines(glider);
```

Then, the splines are used to transfer the normal modes from the solid to the aerodynamic mesh.

```
# Transfer normal modes to aerodynamic model

  glidera = mfs_transfer(glider, gliders, "modes", "disp");
  mfs_export("aero.modes", "msh", glidera, "modes", "disp");
```

Finally, the solid and the aerodynamic mesh files are combined into file glider.msh, and the files containing the normal modes are combined into

file `glider.modes`.

```
# Combine models and modes

  mfs_merge("solid.msh", "aero.msh", "glider.msh", "msh");
  mfs_merge("solid.modes", "aero.modes", "glider.modes", "msh");

  fclose(fid);
```

The correct definition of the aerodynamic mesh can be checked by looking at the combined mesh. In addition, the correct definition of the splines can be checked by looking at the combined mode shapes. Figure 5.1-7 shows the combined shape of mode 14, the highest mode of interest. It can be seen that the deformation normal to the lifting surfaces is correctly transferred from the solid to the aerodynamic mesh.

Gust Response

The calculation of the gust response comprises three steps. First, the transfer functions are computed. Next, the Fourier transform of the responses is obtained by multiplying the transfer functions with the Fourier transform of the gust profile. Finally, an inverse Fourier transform yields the transient response.

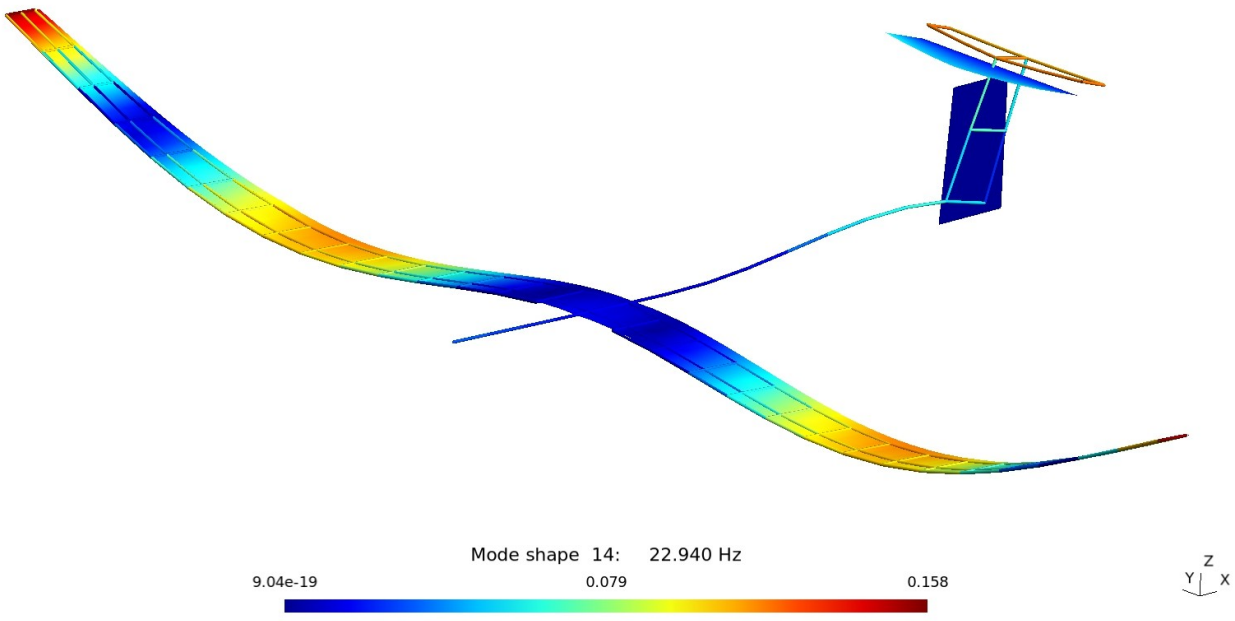


Figure 5.1-7: Mode 14: Combined Solid and Aerodynamic Mesh

The files to compute the gust response are located in directory `gust`. GNU Octave script `trf.m` computes the transfer functions. The transfer functions describe the frequency response to a harmonic gust with unit amplitude. The script begins with the definition of some data.

```
# Example: Frequency response analysis of a standard class glider
#          Transfer functions for gust response
#          Files needed:  solid.bin, aero.bin, gust.bin
#          Files created: trf.bin
# ------------------------------------------------------------

  addpath("../models");    % Path to model definition

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

# Data (kg, m, s)

  rho =     1.21;   % Mass density of the air
  w0  =        1;   % Gust amplitude
  v   = [30, 60];   % Flight velocities
  fc  =       10;   % Cut-off frequency
```

The transfer functions are computed for excitation frequencies up to the cut-off frequency. To make subsequent computations easier, we use the same frequency increment as for the computation of the Fourier transform. However, because the glider is not supported, we cannot compute transfer functions at 0 Hz.

```
# Excitation frequencies

  load ../loads/gust.bin
  f    = df : df : fc;
```

Next, we define the aeroelastic model. First, the model type, the solid and the aerodynamic component and the splines are defined.

```
# Define aeroelastic model

  load ../models/solid.bin
  load ../models/aero.bin

  model = struct("type", "aeroelastic",
                 "solid", gliders, "aero", glidera,
                 "splines", splinedef());
```

Then, we define two different harmonic gusts for the two different flight velocities. Both gusts are given a unit amplitude.

```
# Define gusts
```

```
  qdyn = mat2cell(0.5 * rho * v.^2, 1, [1, 1]);

  model.loads.gust = struct("wg",   w0,
                            "v" ,   mat2cell(v, 1, [1, 1]),
                            "qdyn", qdyn,
                            "lc",   {1, 2});
```

Then, the aeroelastic component is created and the splines are computed.

```
# Build aeroelastic component and compute splines

  glider = mfs_new(fid, model);
  glider = mfs_splines(glider);
```

Finally, the transfer functions are computed and the aeroelastic component is saved in file `trf.bin`. In an aeroelastic frequency response analysis, the default method used by function **`mfs_freqresp`** is a modal frequency response analysis improved by the mode acceleration technique. Also, by default, no additional excitation frequencies are added in the vicinity of the resonance frequencies because the actual resonance frequencies are altered by the aerodynamic stiffness. Since we use these defaults, we only need to specify the load case.

```
# Compute the transfer functions

  glider = mfs_freqresp(glider, f, "loadcase", 1);
  glider = mfs_freqresp(glider, f, "loadcase", 2);

  save -binary trf.bin glider

  fclose(fid);
```

After completing the frequency response analysis, we can use GNU Octave script `trf_plot.m` to plot the transfer functions. First, we define the response degrees of freedom. Then we load the aeroelastic component that contains the results of the frequency response analysis.

```
# Example: Frequency response analysis of a standard class glider
#          Plot the transfer functions for the gust response
#          Files needed:  trf.bin
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

# Define response degrees of freedom and wing root element

  rid   = [5, 3; 15, 3; 115, 3];   % Node, Dof

# Load the aeroelastic component
```

```
load trf.bin
```

As we are interested in results of the solid component, we first extract the solid component from the aeroelastic component. Then, we use function **mfs_getset** to get the identifiers of the elements of the main spar of the left wing. The first of these elements is the element at the wing root. Finally we can use function **mfs_getresp** to get the desired results. The data are collected in cell arrays.

```
# Extract the solid component

  gliders = mfs_extract(glider, "solid");
  clear glider

# Get identifier of element at wing root

  eltids = mfs_getset(gliders, "eset", "left_spar");
  eltid  = eltids(1);

# Extract the transfer functions

  f   = mfs_getresp(gliders, "freqresp", "freq");
  for lc = 1 : 2
      U{lc}  = mfs_getresp(gliders, "freqresp", "disp", rid, lc);
      UR{lc} = mfs_getresp(gliders, "freqresp", "reldisp",
                           rid, lc);
      A{lc}  = mfs_getresp(gliders, "freqresp", "acce", rid, lc);
```

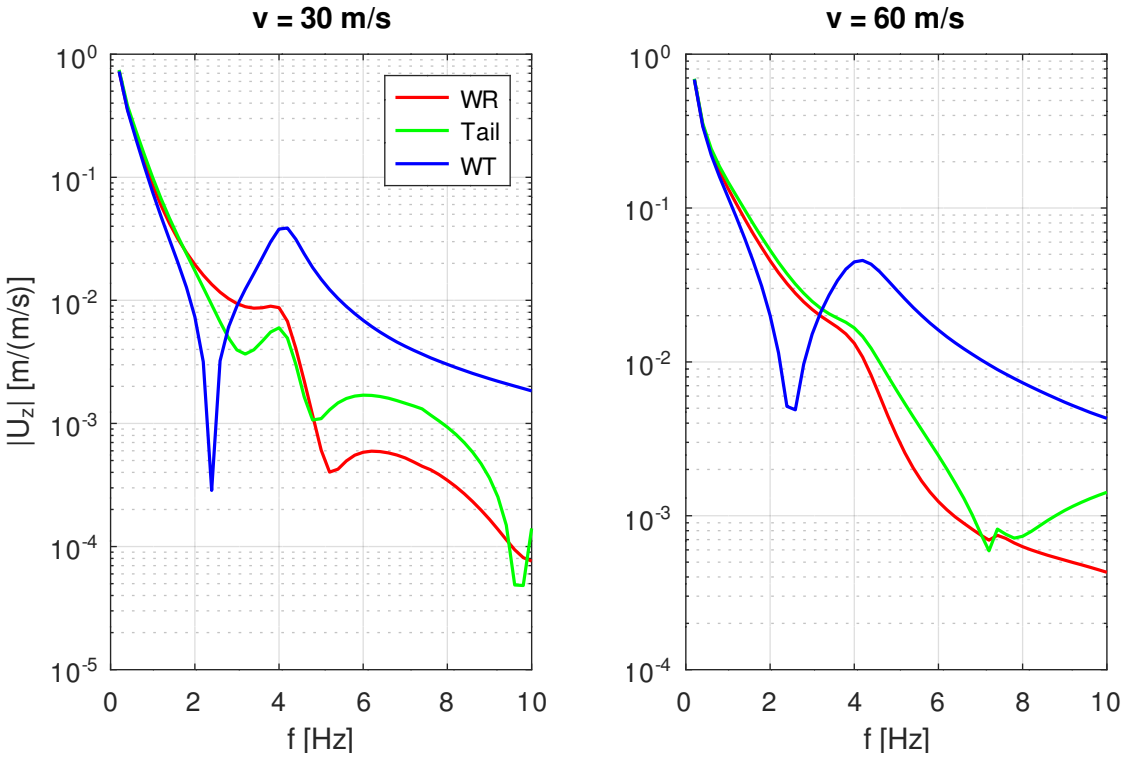


Figure 5.1-8: Gust: Transfer Functions of Absolute Displacements

```
    R      = mfs_getresp(gliders, "freqresp", "resultant",
                         eltid, lc);
    My{lc} = R{1}.My;
endfor
```

Subsequently, the results can be plotted.

```
# Plot the transfer functions

  figure(1, "position", [100, 200, 800, 500],
            "paperposition", [0, 0, 17, 10]);
  subplot(1, 2, 1);
     semilogy(f, abs(U{1}));
     legend("WR", "Tail", "WT");
     title("v = 30 m/s");
     grid;
     ylabel("|U_z| [m/(m/s)]"); xlabel("f [Hz]");
  subplot(1, 2, 2);
     semilogy(f, abs(U{2}));
     title("v = 60 m/s");
     grid;
     xlabel("f [Hz]");
  print(["TF_U", EXT], FORMAT);

  figure(2, "position", [400, 200, 800, 500],
            "paperposition", [0, 0, 17, 10]);
  subplot(1, 2, 1);
     semilogy(f, abs(UR{1}));
     legend("WR", "Tail", "WT", "location", "southwest");
```

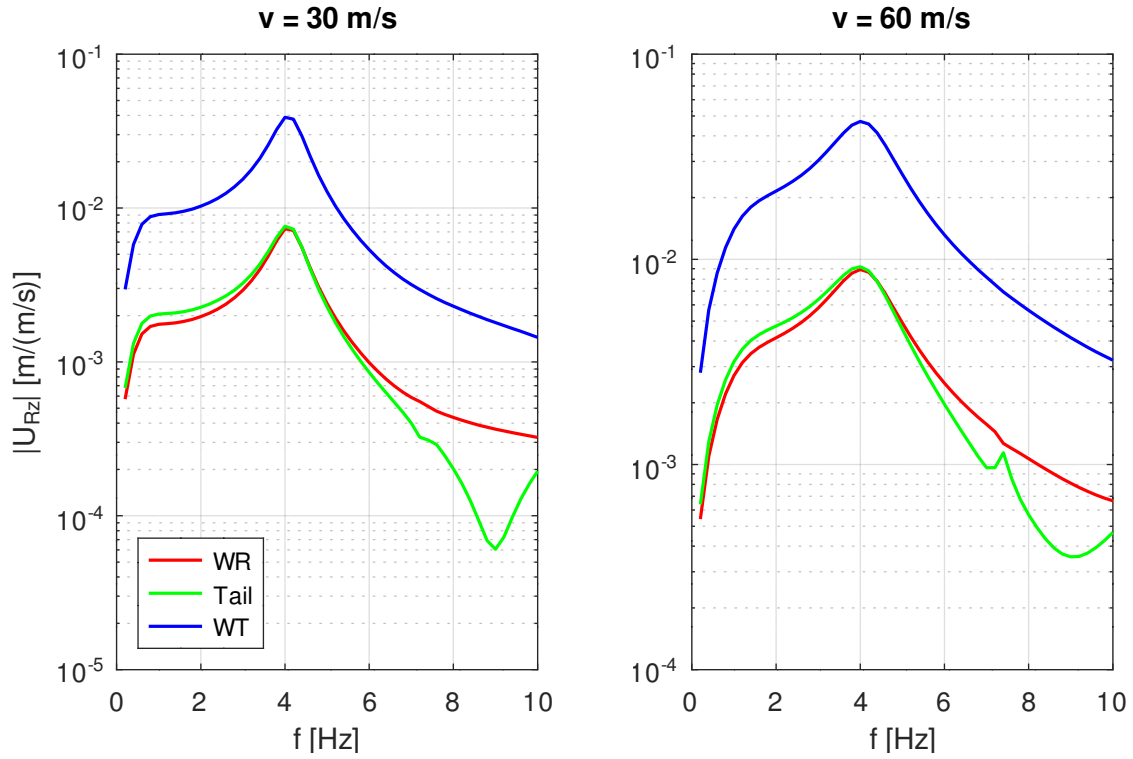


Figure 5.1-9: Gust: Transfer Functions of Relative Displacements

```
   title("v = 30 m/s");
   grid;
   ylabel("|U_{Rz}| [m/(m/s)]"); xlabel("f [Hz]");
subplot(1, 2, 2);
   semilogy(f, abs(UR{2}));
   title("v = 60 m/s");
   grid;
   xlabel("f [Hz]");
print(["TF_A", EXT], FORMAT);

figure(3, "position", [700, 200, 800, 500],
          "paperposition", [0, 0, 17, 10]);
subplot(1, 2, 1);
   semilogy(f, abs(A{1}));
   legend("WR", "Tail", "WT", "location", "south");
   title("v = 30 m/s");
   grid;
   ylabel("|A_z| [(m/s^2)/(m/s)]"); xlabel("f [Hz]");
subplot(1, 2, 2);
   semilogy(f, abs(A{2}));
   title("v = 60 m/s");
   grid;
   xlabel("f [Hz]");
print(["TF_A", EXT], FORMAT);

figure(4, "position", [1000, 200, 800, 500],
          "paperposition", [0, 0, 17, 10]);
subplot(1, 2, 1);
   semilogy(f, abs(My{1}));
   title("v = 30 m/s");
```

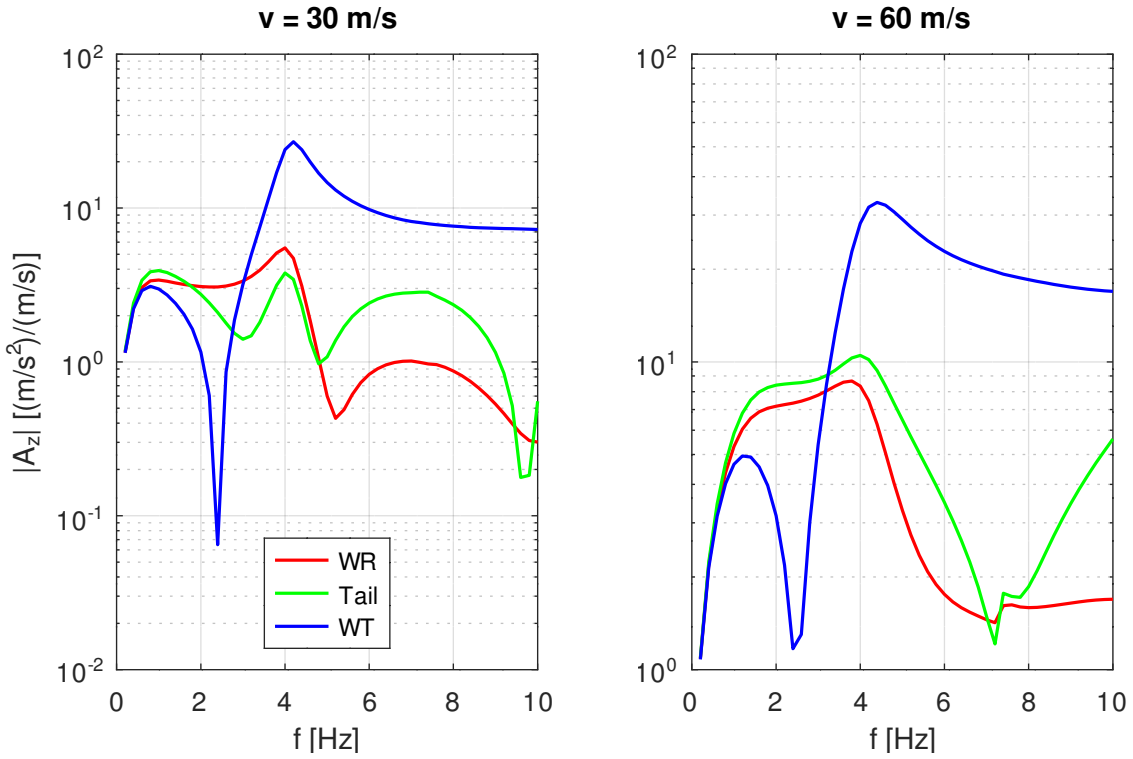


Figure 5.1-10: Gust: Transfer Functions of Accelerations

```
    grid;
    ylabel("|M_y| [Nm/(m/s)]");
    xlabel("f [Hz]");
subplot(1, 2, 2);
    semilogy(f, abs(My{2}));
    title("v = 60 m/s");
    grid;
    xlabel("f [Hz]");
print(["TF_My", EXT], FORMAT);
```

The script generates four diagrams, each containing two sets of transfer functions for the two flight velocities. Figures 5.1-8 to 5.1-10 show the transfer functions of the absolute displacements, the displacements relative to the rigid body motion and the accelerations. WR denotes the wing root and WT the wing tip. It can be seen that the transfer functions of the absolute displacements become infinitely large at a frequency of 0 Hz. The transfer functions of the relative displacements and of the accelerations are zero at 0 Hz.

Figure 5.1-11 shows the transfer functions of the wing root bending moment. These transfer functions are also zero at 0 Hz.

GNU Octave script `response.m` computes the Fourier transform of the response and the transient response.

First, we define the response degrees of freedom as well as the time step for the transient response and the duration of the transient response to be plotted.

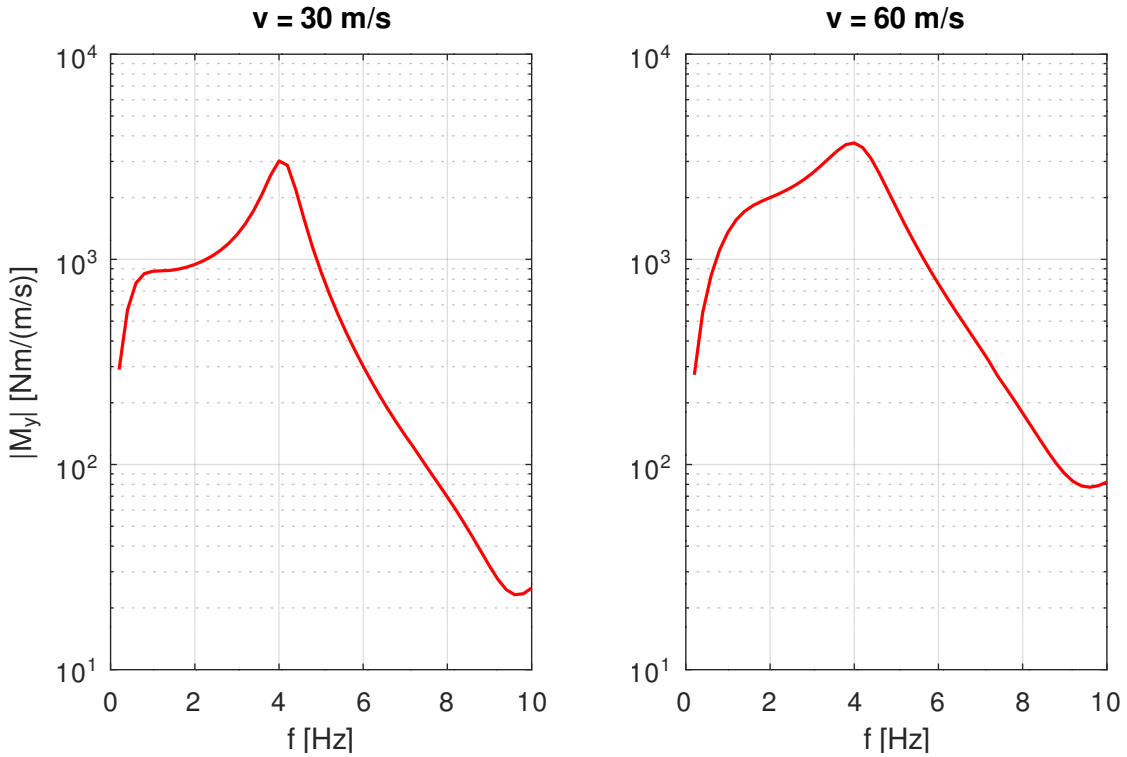


Figure 5.1-11: Gust: Transfer Functions of Wing Root Bending Moment

```
# Example: Frequency response analysis of a standard class glider
#          Gust response
#          Files needed:  trf.bin, gust.bin
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

# Define response degrees of freedom

  rid   = [5, 3; 15, 3; 115, 3];   % Node, Dof

# Define parameters of transient response

  dt = 0.005;   % Time step
  te =     3;   % Duration (for plotting)
```

Next, we load the aeroelastic component, extract the solid component, get the identifier of the element at the wing root and access the frequencies. For the inverse Fourier transform to work correctly we must include the zero frequency. Then we load the Fourier transforms of the gusts.

```
# Get solid component with frequency response

  load trf.bin
  gliders = mfs_extract(glider, "solid");
  clear glider

# Get identifier of element at wing root

  eltids = mfs_getset(gliders, "eset", "left_spar");
  eltid  = eltids(1);

# Get frequencies, include zero frequency

  f  = [0, mfs_getresp(gliders, "freqresp", "freq")];
  nf = length(f);

# Get Fourier transforms of the gusts

  load ../loads/gust.bin
```

The responses are computed in a nested loop. The outer loop is over the two velocities and the inner over the two gusts. The transfer functions depend only on the flight velocity. Therefore, we can access them in the outer loop. The transfer functions of the first load case belong to the first flight velocity and those of the second load case to the second. The values of the transfer functions at the zero frequency are set to zero. Of course, this is not correct for the transfer functions of the absolute displacements.

```
# Compute responses

  for lc = 1 : 2        % Velocity loop

#   Get transfer functions (values at f = 0 are zero)

      TU(:, 2 : nf)  = mfs_getresp(gliders, "freqresp",
                                   "disp", rid, lc);
      TUR(:, 2 : nf) = mfs_getresp(gliders, "freqresp",
                                   "reldisp", rid, lc);
      TA(:, 2 : nf)  = mfs_getresp(gliders, "freqresp",
                                   "acce", rid, lc);
      R              = mfs_getresp(gliders, "freqresp",
                                   "resultant", eltid, lc);
      TM(2 : nf)     = R{1}.My;

      for k = 1 : 2    % Gust loop
```

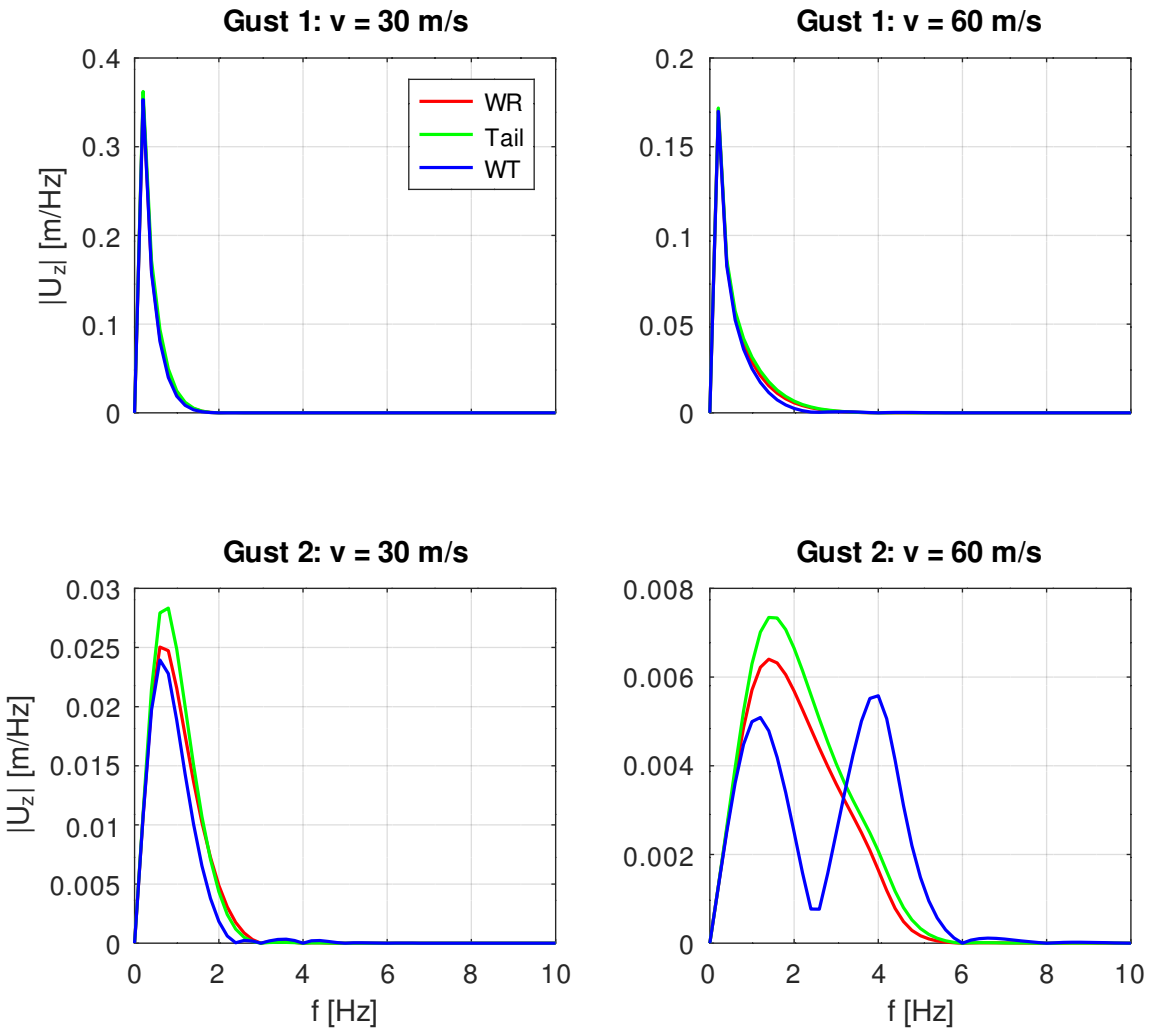


Figure 5.1-12: Gust: Fourier Transforms of Absolute Displacements

```
            titles{lc, k} = sprintf("Gust %1d: v = %2d m/s",
                                    k, v(lc));
```

The Fourier transforms of the responses are obtained by multiplying the transfer functions with the Fourier transforms of the gusts.

```
#       Compute Fourier transforms of responses

            WG = W{k}(lc, 1 : nf);
            U{lc, k} = TU .* WG; UR{lc, k} = TUR .* WG;
            A{lc, k} = TA .* WG; M{lc, k}  = TM .* WG;
```

Subsequently, we use function **`mfs_freq2time`** to compute the transient responses from their Fourier transforms. The results are collected in cell arrays to be plotted later. The first argument of function **`mfs_freq2time`** is an array

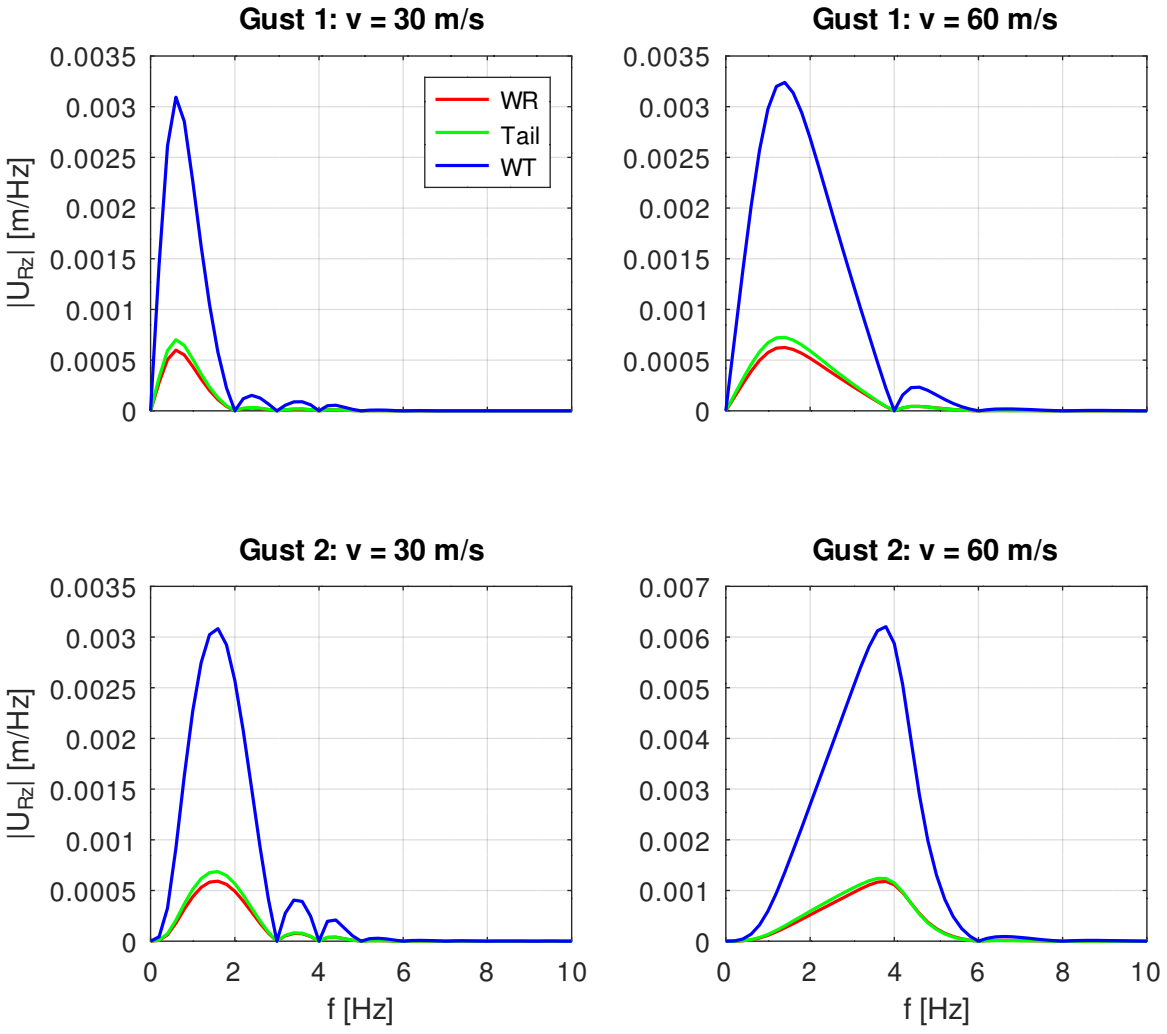


Figure 5.1-13: Gust: Fourier Transforms of Relative Displacements

with the values of the Fourier transform corresponding to the positive frequencies. The second argument is the frequency increment. These two arguments are required. The third argument defines the time step of the transient response. This argument is optional. If it is omitted, a time step is determined from the frequencies.

The fourth argument controls the base line correction. If it is omitted or is zero, no correction is applied. With a value of 2, a linear function is added to the transient response so that its value as well as the value of its first derivative at $t = 0$ s are zero. For the first gust, the absolute displacements are computed first without correction and then with correction.

```
#       Compute transient responses

           [u{lc, k}, t] = mfs_freq2time(U{lc, k}, df, dt);
           uR{lc, k}     = mfs_freq2time(UR{lc, k}, df, dt);
```

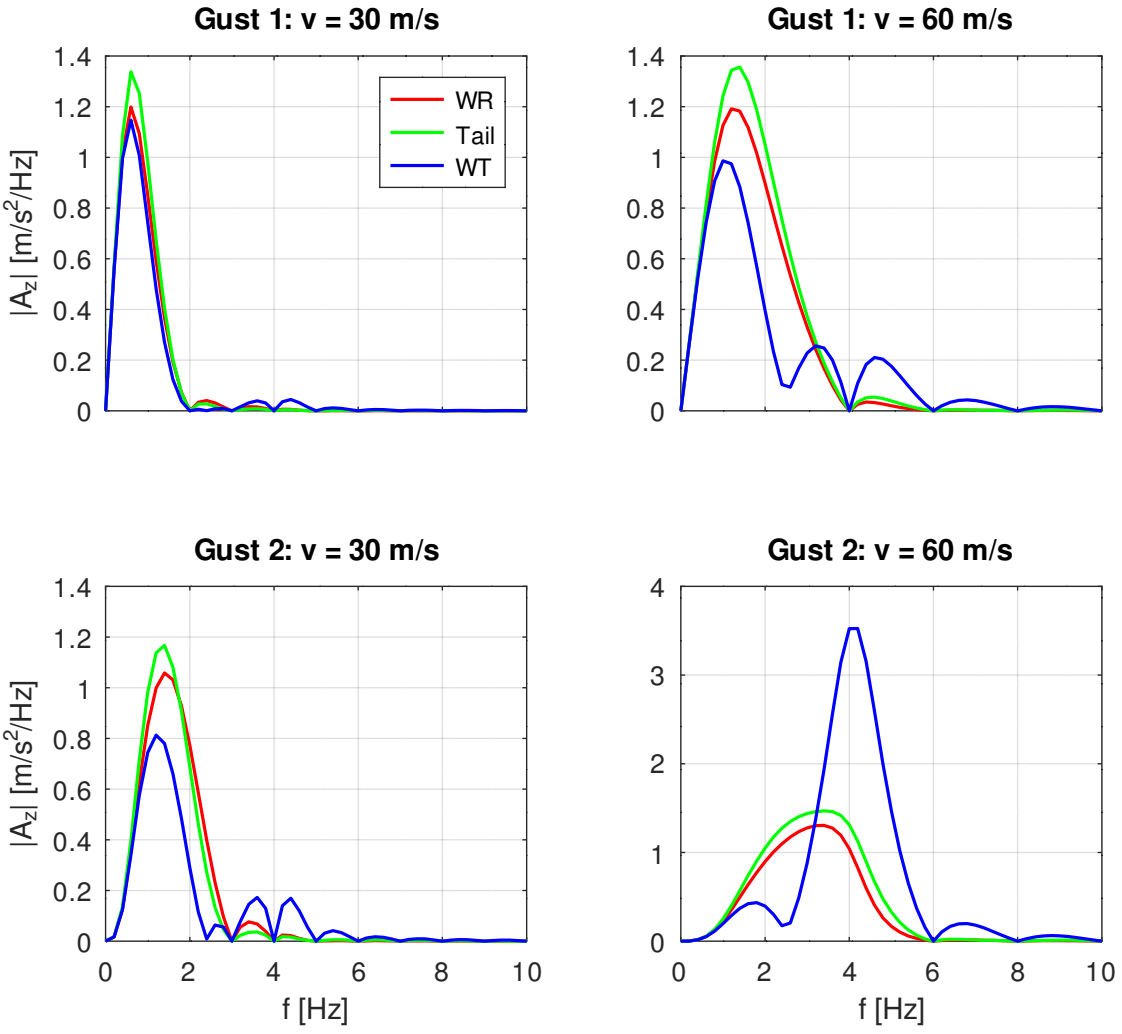


Figure 5.1-14: Gust: Fourier Transforms of Accelerations

```
          a{lc, k}      = mfs_freq2time(A{lc, k}, df, dt);
          m{lc, k}      = mfs_freq2time(M{lc, k}, df, dt);

          if (k == 1)
             uc{lc} = mfs_freq2time(U{lc, k}, df, dt, 2);
          endif

      endfor            % Gust loop

  endfor                % Velocity loop
```

Finally, the displacements are saved and the results plotted.

```
# Save transient displacements for later comparison

  save -binary response.bin t u

# Plot Fourier transforms of responses

  % Displacements

  figure(1, "position", [100, 100, 800, 800],
            "paperposition", [0, 0, 17, 15]);
  subplot(2, 2, 1)
     plot(f, abs(U{1, 1}));
     legend("WR", "Tail", "WT");
     title(titles{1, 1});
     grid;
     axis("labely");
```

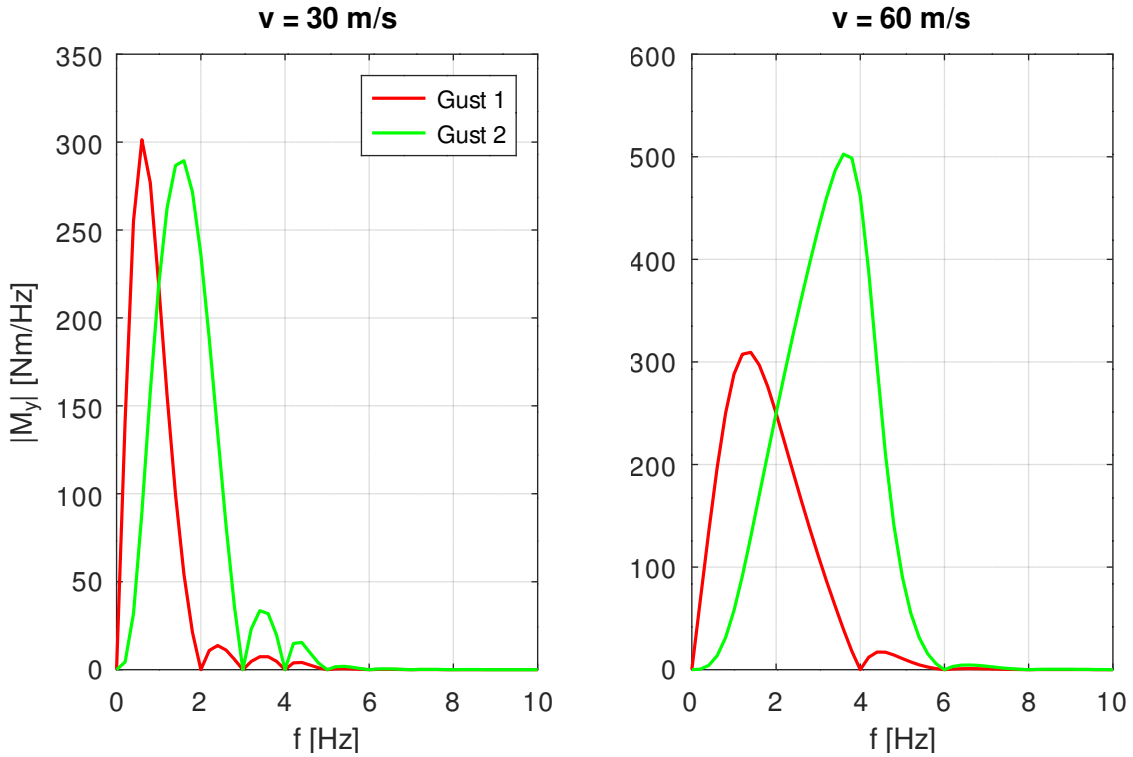


Figure 5.1-15: Gust: Fourier Transforms of Wing Root Bending Moment

```
   ylabel("|U_z| [m/Hz]");
subplot(2, 2, 2)
   plot(f, abs(U{2, 1}));
   title(titles{2, 1});
   grid;
   axis("labely");
subplot(2, 2, 3)
   plot(f, abs(U{1, 2}));
   title(titles{1, 2});
   grid;
   ylabel("|U_z| [m/Hz]"); xlabel("f [Hz]");
subplot(2, 2, 4)
   plot(f, abs(U{2, 2}));
   title(titles{2, 2});
   grid;
   xlabel("f [Hz]");
print(["FT_U", EXT], FORMAT);
      …
```

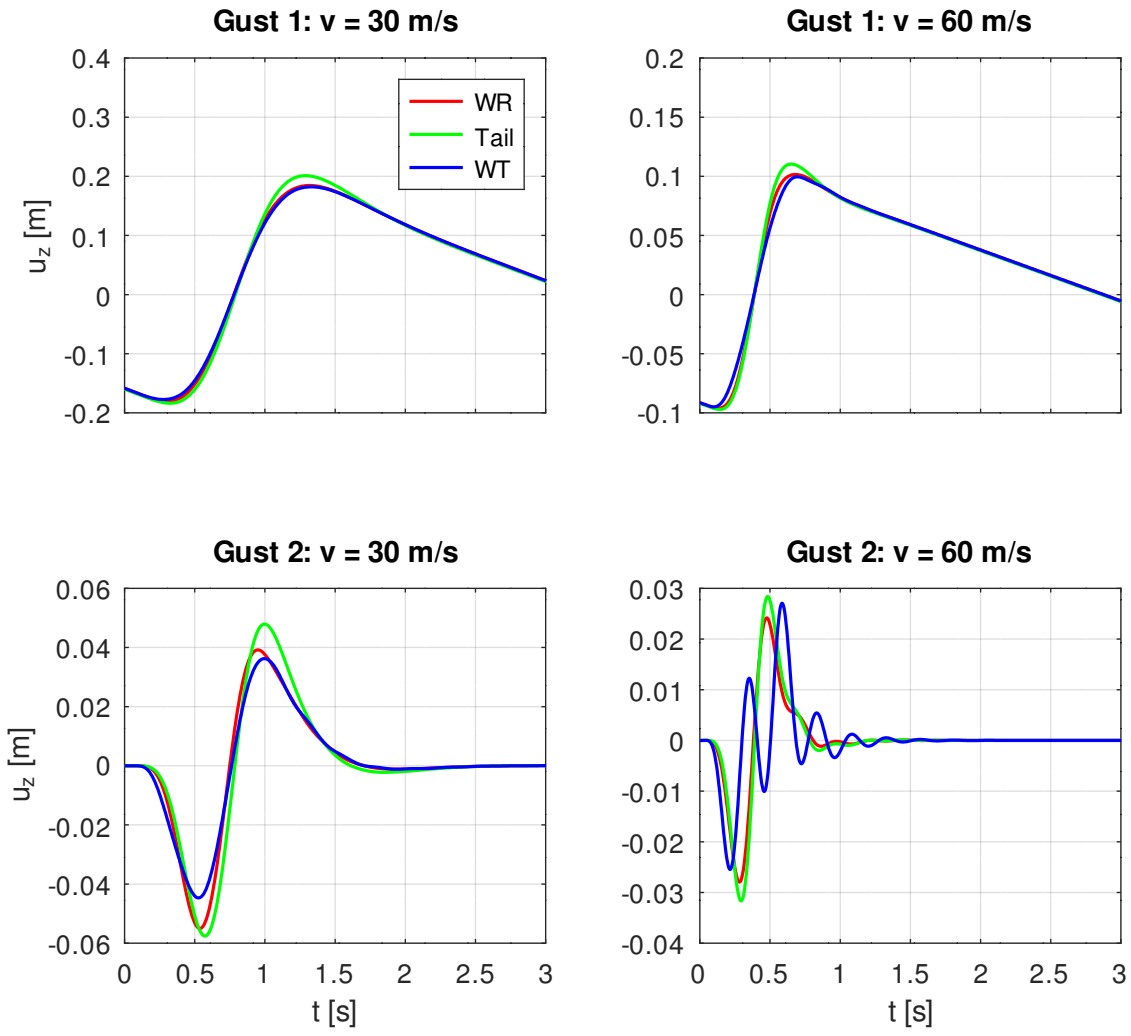


Figure 5.1-16: Gust: Absolute Displacements (without Correction)

Figure 5.1-12 shows the Fourier transforms of the absolute displacements. For gust 1, like the corresponding transfer functions, the Fourier transforms go to infinity at 0 Hz. This is an indication that the Fourier transforms of the absolute displacements do not exist. In fact, the vertical displacement of the glider does not return to zero after passing gust 1, but the glider continues its flight with a constant gain in altitude. As the value of the Fourier transform at 0 Hz equals the area under the function that is transformed, this value is in fact infinitely large.

For gust 2, the absolute displacements go to zero at 0 Hz. The fact that the Fourier transform of the second gust profile is zero at 0 Hz cures the problem that the transfer functions of the absolute displacements are infinitely large at 0 Hz. Actually, after passing gust 2, the glider returns to the same altitude as before so that the Fourier transform of the absolute vertical displacements exists.

Figures 5.1-13 to 5.1-15 show the Fourier transforms of the relative displacements, the accelerations and the bending moment at the wing root. The corresponding transfer functions are all zero at 0 Hz, and so are the Fourier transforms of the responses. All these responses return to zero some time after the glider has passed the gust. Thus, the Fourier transforms should exist.

Figure 5.1-16 shows the absolute displacements computed without baseline correction. It can be seen that the displacements due to gust 1 do not start at

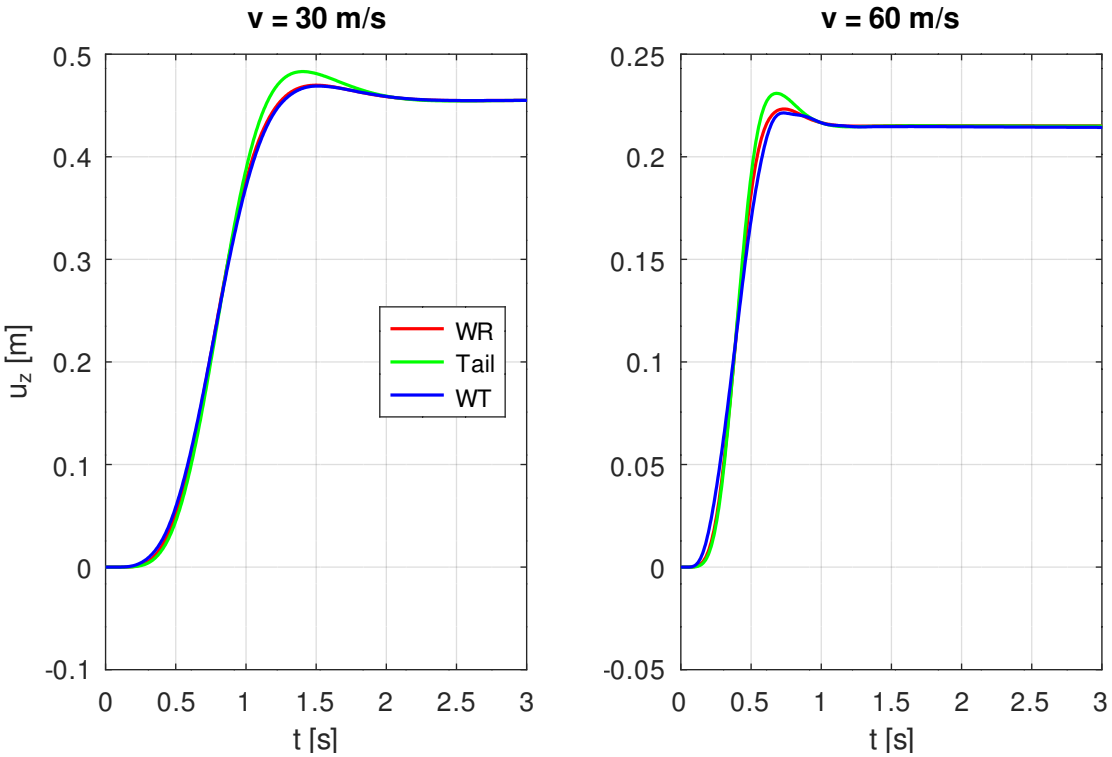


Figure 5.1-17: Gust 1: Absolute Displacements (with Correction)

zero but at some negative value. The first derivatives also do not start at zero and do not go to zero with increasing time. This physically impossible effect is due to the fact that the inverse Fourier transform lacks the infinite contribution of the zero frequency. The absolute displacements due to gust 2, on the other hand, are correct.

Figure 5.1-17 shows the absolute displacements due to gust 1 computed with baseline correction. It can be seen that the baseline correction cures the problem and leads to physically meaningful results.

Figures 5.1-18 to 5.1-20 show the relative displacements, the accelerations and the bending moment at the wing root. All these responses start with zero and go towards zero with increasing time.

Finally, we want to animate the motion of the glider when it passes gust 2 at a

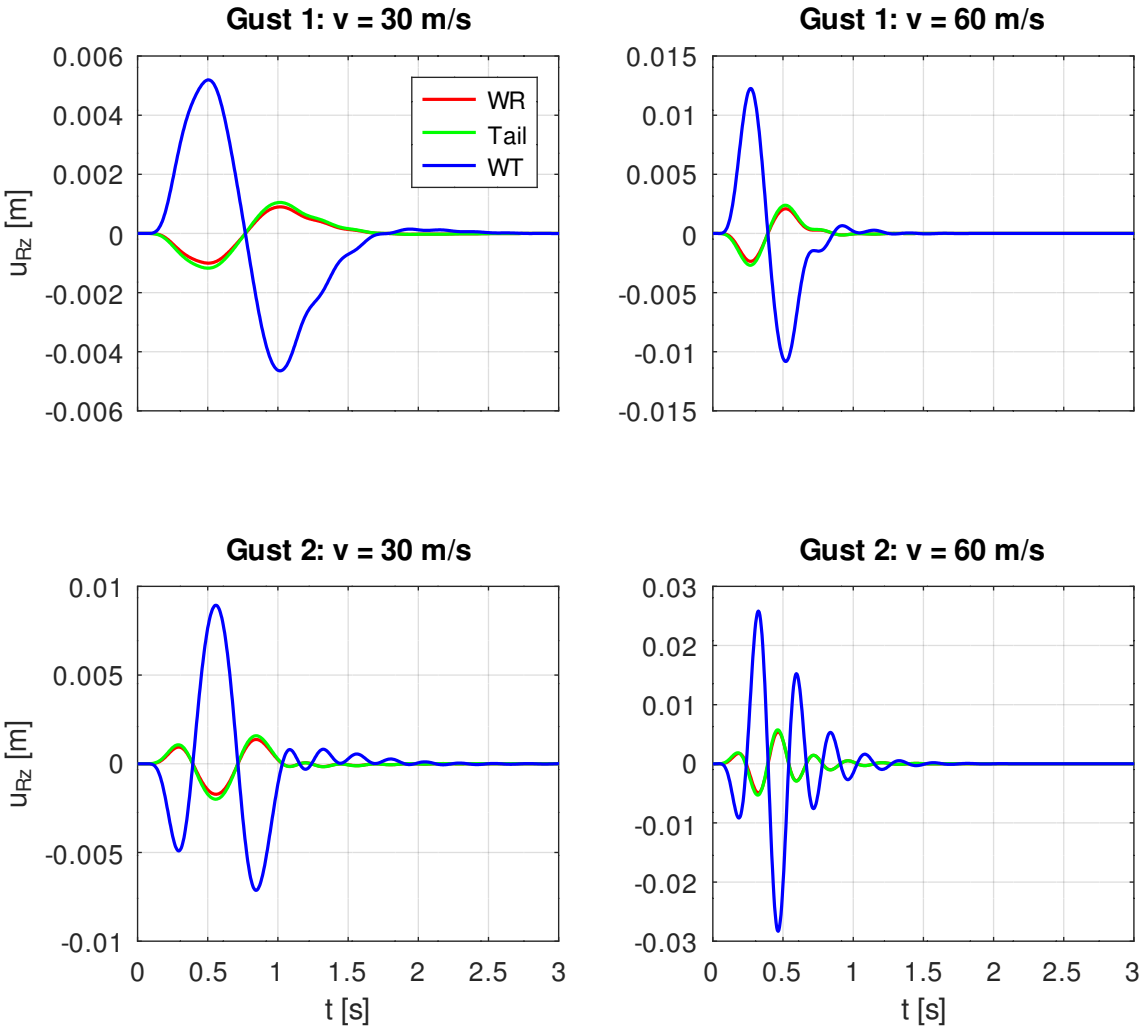


Figure 5.1-18: Gust: Relative Displacements

velocity of $60\ \mathrm{m/s}$. GNU Octave script `motion.m` computes the transient displacements at all nodal points and exports them to Gmsh.

The script begins with the definition of the response degrees of freedom, the time step for the transient response and the duration of the animation. The time step is larger and the duration is smaller than in the previous analysis because the current analysis will generate a large amount of data. The response degrees of freedom are used to compare the transient displacements with those obtained in the previous analysis (script `response.m`).

```
# Example: Frequency response analysis of a standard class glider
#          Gust response: Transform results of second gust to
#                         time domain and export displacements
#                         for movie
#          Files needed:  trf.bin, gust.bin, reponse.bin
# ------------------------------------------------------------
```

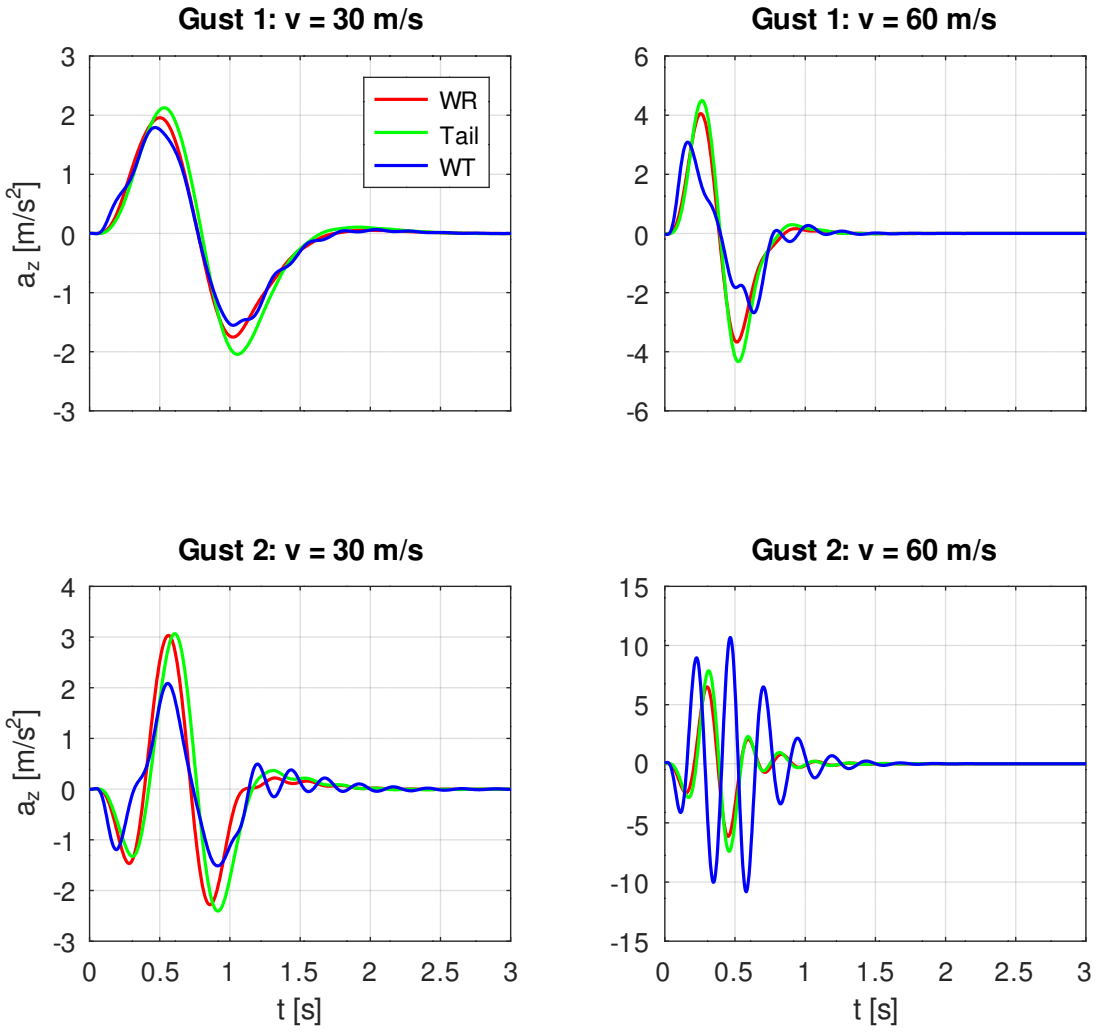


Figure 5.1-19: Gust: Accelerations

```
# Define response degrees of freedom

  rid = [5, 3; 15, 3; 115, 3];    % Node, Dof

# Parameters of transient response

  dt = 0.010;    % Time step for transformation to time domain
  te =     2;    % Time for plotting and animation
```

Then the aeroelastic component is loaded and the solid component is extracted. Afterwards, the Fourier transforms of the gusts are loaded.

```
# Get solid component with frequency response

  load trf.bin
  gliders = mfs_extract(glider, "solid");
  clear glider

# Get Fourier transforms of the gusts

  load ../loads/gust.bin
```

For an animation, the displacements at all nodal points are needed. Therefore, the second form of function **`mfs_freq2time`** is used. In the second form, the function performs the inverse Fourier transform of the modal results and stores them as results of a transient response analysis.

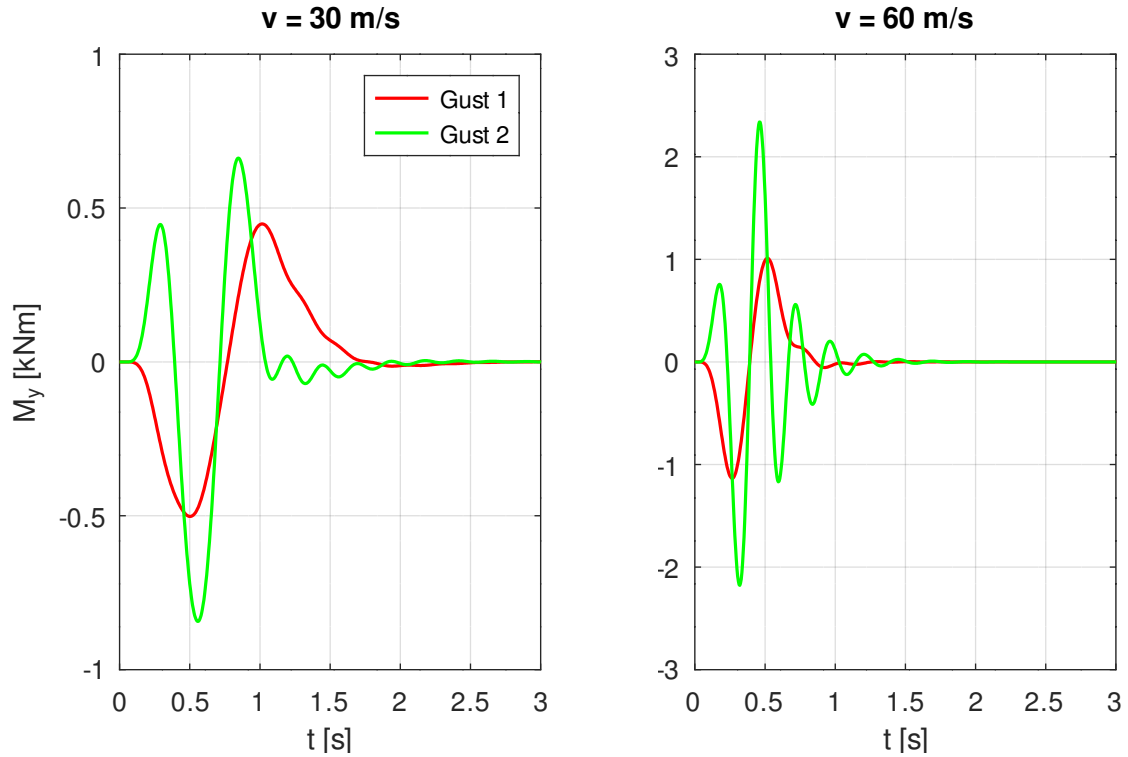


Figure 5.1-20: Gust: Bending Moment at the Wing Root

Structure **spectrum** is used to assign the Fourier transform of the second gust to load case 2 of the frequency response analysis. The function will use these data to compute the Fourier transform of the modal results and to perform the inverse Fourier transform. The results are stored as result case 1 of a transient response analysis.

```
# Inverse Fourier transform for second gust and second velocity

  spectrum = struct("df", df, "lc", 2,
                    "spec", W{2}(2, :));

  gliders = mfs_freq2time(gliders, spectrum, 1, dt);
```

The transient results thus obtained can be accessed in the same way as any results from a transient response analysis. The following commands generate a diagram comparing the displacements with those of the previous analysis. You will see that the results are identical.

```
# Plot time history (for comparison with response.m)

  load response.bin
  ur = u{2, 2};

  tm = mfs_getresp(gliders, "transresp", "time");
  um = mfs_getresp(gliders, "transresp", "disp", rid);

  figure(1, "position", [100, 100, 1000, 800]);
  subplot(3, 1, 1);
     plot(t, ur(1, :), tm, um(1, :))
     title(sprintf("Node %3.0d", rid(1, 1)));
     legend("response", "motion");
     grid;
     xlim([0, te]);
     ylabel('u_z [m]');
  subplot(3, 1, 2);
     plot(t, ur(2, :), tm, um(2, :))
     title(sprintf("Node %3.0d", rid(2, 1)));
     grid;
     xlim([0, te]);
     ylabel('u_z [m]');
  subplot(3, 1, 3);
     plot(t, ur(3, :), tm, um(3, :))
     title(sprintf("Node %3.0d", rid(3, 1)));
     grid;
     xlim([0, te]);
     xlabel("t [s]"); ylabel('u_z [m]');
```

Finally, function **mfs_back** is used to obtain the displacements at all nodal points. The displacements are then exported to Gmsh where they can be animated using the standard animation keys. Figure 5.1-21 shows the deformation at time step 68.

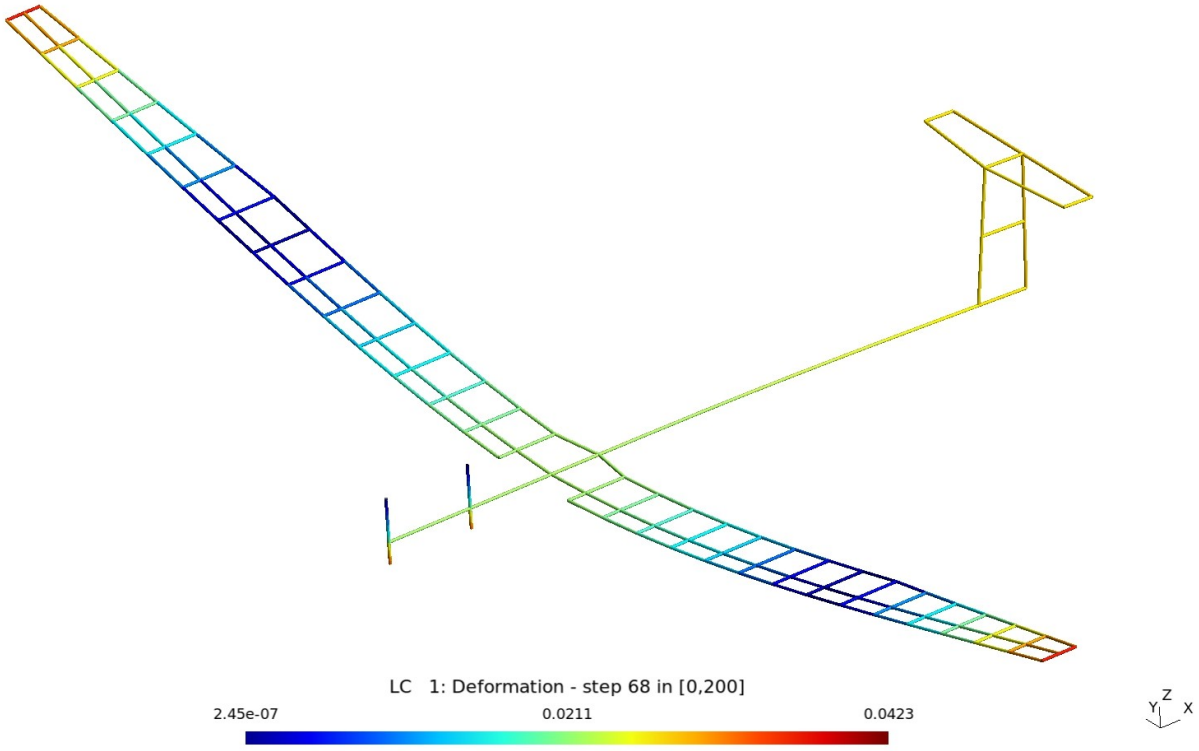


Figure 5.1-21: Gust: Deformation at Time Step 68

```
# Perform backtransformation and export displacements

  tb = t(find(tm <= te));
  gliders = mfs_back(gliders, "transresp", "disp", 1, tb);
  mfs_export("gust2.dsp", "msh", gliders, "transresp", "disp");
```

Response to Elevator Input

The files to compute the response to an elevator input are located in directory `elevator`. The calculations are done in the same way as the calculation of the response to a gust.

First, GNU Octave script `trf.m` computes the transfer functions. The transfer functions describe the frequency response to a harmonic elevator motion with unit amplitude.

The script begins with the definition of the data. Next, the excitation frequencies are defined. Again, the zero frequency must be excluded because the static problem has no unique solution.

```
# Example: Frequency response analysis of a standard class glider
#          Transfer functions for response to elevator input
#          Files needed:  solid.bin, aero.bin, elevator.bin
#          Files created: trf.bin
# ------------------------------------------------------------
```

```
  addpath("../models");    % Path to model definition

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

# Data (kg, m, s)

  rho = 1.21;   % Mass density of the air
  eta =    1;   % Amplitude of elevator motion
  v   =   40;   % Flight velocity
  fc  =   10;   % Cut-off frequency

# Excitation frequencies

  load ../loads/elevator.bin
  f    = df : df : fc;
```

Then we define the aeroelastic model.

```
# Define aeroelastic model

  load ../models/solid.bin
  load ../models/aero.bin

  model = struct("type", "aeroelastic",
                 "solid", gliders, "aero", glidera,
                 "splines", splinedef());
```

In this analysis, there is only one load case. To define the manoeuvre, we specify the dynamic pressure, the flight velocity and a unit amplitude of the harmonic elevator motion.

```
# Define elevator motion

  model.loads.manoeuvre = struct("qdyn", 0.5 * rho * v^2,
                                 "v", v,
                                 "elevator", eta);
```

The commands to perform the analysis are the same as for calculating the gust response. Since we use the default values of all parameters of the frequency response analysis, no parameters need to be specified.

```
# Build aeroelastic component and compute splines

  glider = mfs_new(fid, model);
  glider = mfs_splines(glider);

# Compute the transfer functions

  glider = mfs_freqresp(glider, f);

  save -binary trf.bin glider

  fclose(fid);
```

GNU Octave script `trf_plot` plots the transfer functions.

```
# Example: Frequency response analysis of a standard class glider
#          Plot the transfer functions for the response to an
#          elevator input
#          Files needed:  trf.bin
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

# Define response degrees of freedom and wing root element

  rid   = [5, 3; 15, 3; 115, 3];   % Node, Dof

# Load the aeroelastic component

  load trf.bin

# Extract the solid component

  gliders = mfs_extract(glider, "solid");
  clear glider

  eltid = mfs_getset(gliders, "eset", "left_spar");

# Extract the transfer functions

  f  = mfs_getresp(gliders, "freqresp", "freq");
```

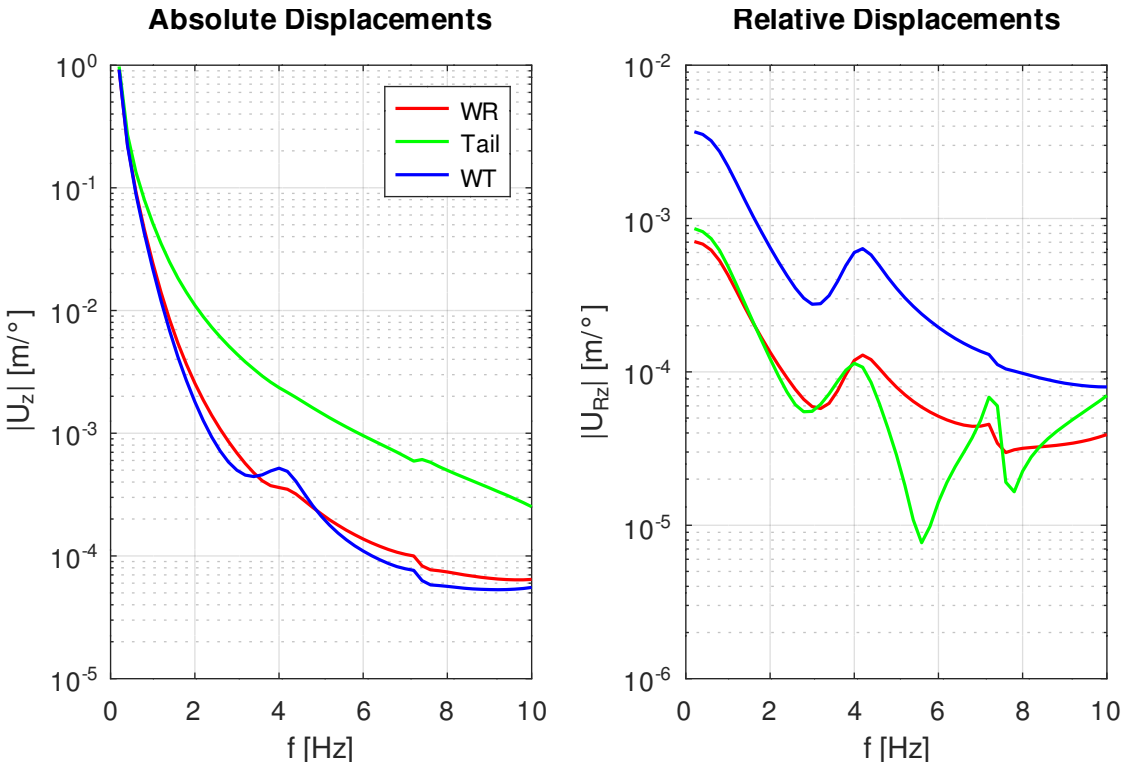


Figure 5.1-22: Elevator Input: Transfer Functions of Displacements

```
  U  = mfs_getresp(gliders, "freqresp", "disp", rid);
  UR = mfs_getresp(gliders, "freqresp", "reldisp", rid);
  A  = mfs_getresp(gliders, "freqresp", "acce", rid);
  R  = mfs_getresp(gliders, "freqresp", "resultant", eltid(1));
  My = R{1}.My;

# Plot the transfer functions

  figure(1, "position", [100, 100, 800, 400],
            "paperposition", [0, 0, 17, 10]);
  subplot(1, 2, 1);
     semilogy(f, abs(U));
     legend("WR", "Tail", "WT");
     title("Absolute Displacements");
     grid;
     ylabel('|U_z| [m/\deg]'); xlabel("f [Hz]");
  subplot(1, 2, 2);
     semilogy(f, abs(UR));
     title("Relative Displacements");
     grid;
     ylabel('|U_{Rz}| [m/\deg]'); xlabel("f [Hz]");
  print(["TF_U", EXT], FORMAT);

  figure(2, "position", [300, 100, 800, 400],
            "paperposition", [0, 0, 17, 10]);
  subplot(1, 2, 1);
     semilogy(f, abs(A));
     legend("WR", "Tail", "WT");
     title("Accelerations");
     grid;
```

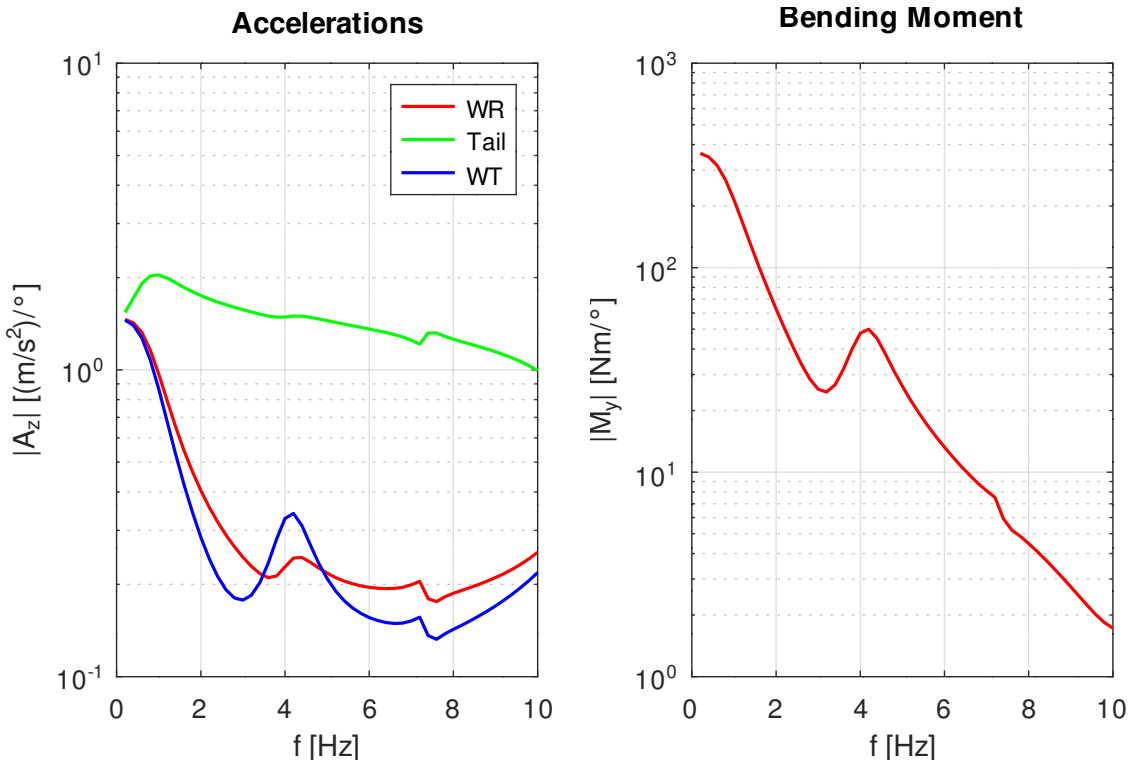


Figure 5.1-23: Elevator Input: Transfer Functions of Accelerations and Bending Moment

```
    ylabel('|A_z| [(m/s^2)/\deg]'); xlabel("f [Hz]");
subplot(1, 2, 2);
    semilogy(f, abs(My));
    title("Bending Moment");
    grid;
    ylabel('|M_y| [Nm/°]'); xlabel("f [Hz]");
print(["TF_AM", EXT], FORMAT);
```

Figure 5.1-22 shows the transfer functions of the absolute and relative displacements. Again, the transfer functions of the absolute displacements go towards infinity when the frequency goes towards zero. The transfer functions of the relative displacements remain finite. Figure 5.1-23 shows the transfer functions of the accelerations and the bending moment at the wing root.

Finally, GNU Octave script `response.m` computes the transient response from the transfer functions and the Fourier transform of the elevator input. This is achieved by using the second form of function **`mfs_freq2time`**. The script also computes the transient displacements at all nodal points and exports them to Gmsh so that they can be animated.

The first part of the script is identical to the script used to compute the response to the gust, but a larger time step is selected.

```
# Example: Frequency response analysis of a standard class glider
#          Response to elevator input
#          Files needed:  trf.bin, elevator.bin
# ------------------------------------------------------------

  [EXT, FORMAT] = iniplot();

  set(0, "defaultaxesfontsize", 10);

# Define response degrees of freedom

  rid   = [5, 3; 15, 3; 115, 3];   % Node, Dof

# Define parameters of transient response

  dt = 0.010;   % Time step
  te =     3;   % Duration (for plotting)

# Get solid component with frequency response

  load trf.bin
  gliders = mfs_extract(glider, "solid");
  clear glider

  eltid = mfs_getset(gliders, "eset", "left_spar");
```

Then the Fourier transform of the elevator input is loaded and assigned to load case 1 of the frequency response analysis. Subsequently, the inverse Fourier transform is performed. Because the glider does not return to its initial

altitude, a baseline correction is applied.

```
# Get Fourier transform of elevator input

  load ../loads/elevator.bin

# Inverse Fourier transform for second gust and second velocity

  spectrum = struct("df", df, "lc", 1, "spec", ETA);

  gliders = mfs_freq2time(gliders, spectrum, 1, dt, 2);
```

Subsequently, we can retrieve and plot desired the results.

```
# Get results

  t  = mfs_getresp(gliders, "transresp", "time");
  u  = mfs_getresp(gliders, "transresp", "disp", rid);
  uR = mfs_getresp(gliders, "transresp", "reldisp", rid);
  a  = mfs_getresp(gliders, "transresp", "acce", rid);
  R  = mfs_getresp(gliders, "transresp", "resultant", eltid(1));
  My = R{1}.My;

# Plot results

  figure(1, "position", [100, 100, 800, 400],
            "paperposition", [0, 0, 17, 10]);
  subplot(1, 2, 1);
     plot(t, u);
```

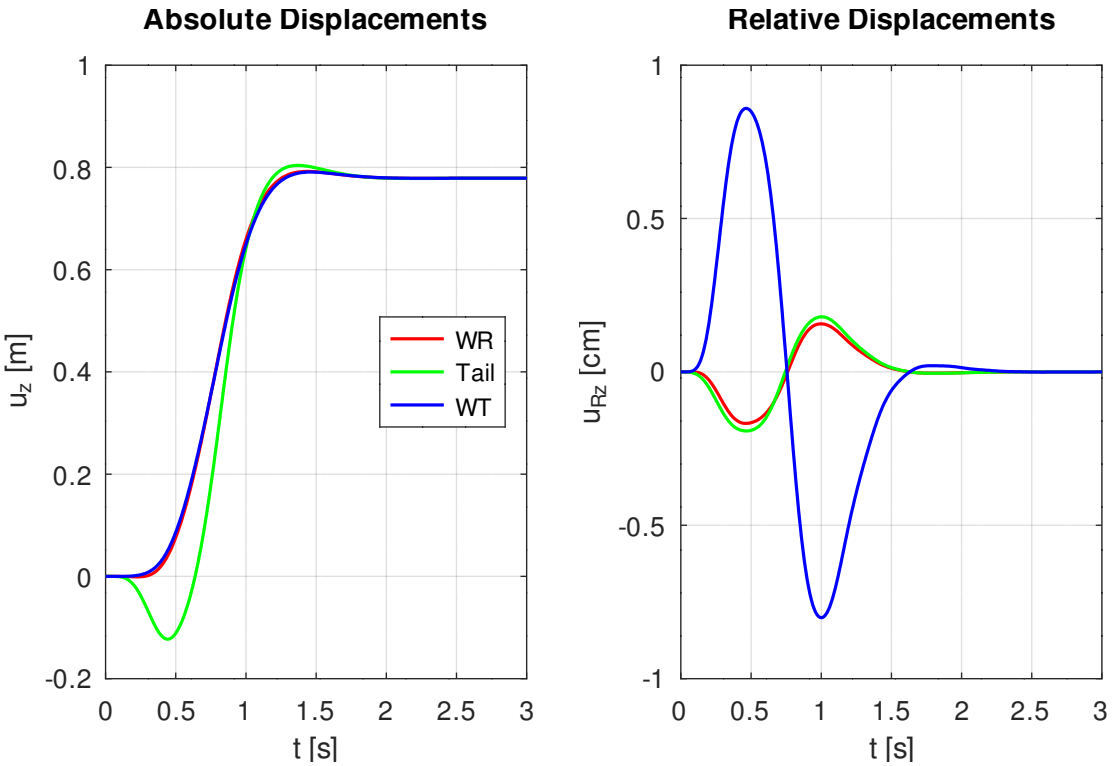


Figure 5.1-24: Elevator Input: Displacements

```
   legend("WR", "Tail", "WT", "location", "east");
   title("Absolute Displacements");
   grid;
   xlim([0, te]);
   ylabel('u_z [m]'); xlabel("t [s]");
subplot(1, 2, 2);
   plot(t, uR * 100);
   title("Relative Displacements");
   grid;
   xlim([0, te]);
   ylabel('u_{Rz} [cm]'); xlabel("t [s]");
print(["u", EXT], FORMAT);

figure(2, "position", [300, 100, 800, 400],
          "paperposition", [0, 0, 17, 10]);
subplot(1, 2, 1);
   plot(t, a);
   legend("WR", "Tail", "WT");
   title("Accelerations");
   grid;
   xlim([0, te]);
   ylabel('a_z [m/s^2]'); xlabel("t [s]");
subplot(1, 2, 2);
   plot(t, My / 1000);
   title("Bending Moment");
   grid;
   xlim([0, te]);
   ylabel('M_y [kNm]'); xlabel("t [s]");
print(["aM", EXT], FORMAT);
```

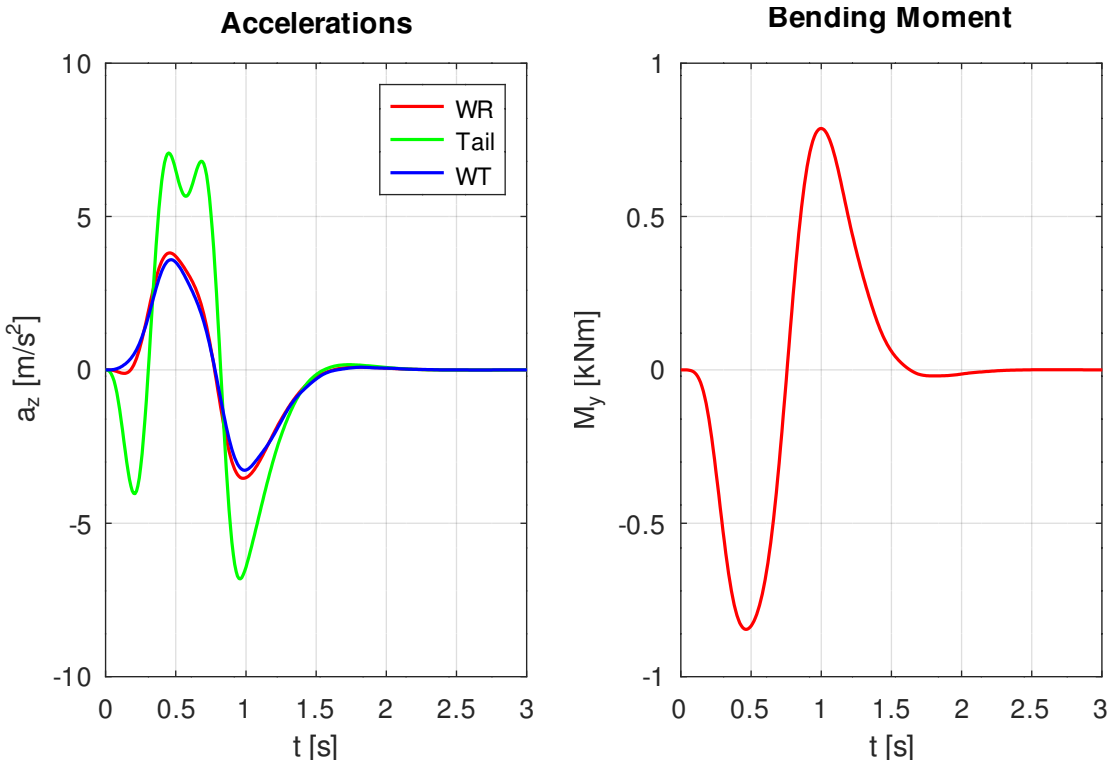


Figure 5.1-25: Elevator Input: Accelerations and Bending Moment

Finally, we compute the displacements at all nodal points and export them to Gmsh.

```
# Perform backtransformation and export displacements

  tb = t(find(t <= te));
  gliders = mfs_back(gliders, "transresp", "disp", 1, tb);
  mfs_export("elevator.dsp", "msh", gliders,
             "transresp", "disp");
```

Figure 5.1-24 shows the absolute and the relative displacements. The absolute displacements do not return to zero but the displacements relative to the rigid body motion do. Please note that the unit of the relative displacements is cm, so they are smaller than the absolute displacements by a factor of 100.

Figure 5.1-25 shows the accelerations and the bending moment at the wing root. Both results return to zero some time after the elevator input.

Both the absolute displacements and the accelerations clearly indicate a rotation of the glider about its transverse axis, as is to be expected for an elevator input.