Contents

A Introduction to the appendices 34
  A.1 Object-oriented design for more flexible software . . . . . . . . . . . . . . . . . . 34
  A.2 Organization of the files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

B Concrete slab 38
  B.1 Running the FEMU software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
  B.2 Simple FEM updating example with two groups of elements . . . . . . . . . . 38
  B.3 Regularization with shape functions . . . . . . . . . . . . . . . . . . . . . . . . . 41
  B.4 Regularized damage detection example . . . . . . . . . . . . . . . . . . . . . . . 43

C Concrete railway arch bridge (Långforsen, Kalix river) 45
  C.1 Measurements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
  C.2 Finite element models and preliminary FEMU results . . . . . . . . . . . . . . . 47

D Steel truss bridge over Åby river 49

E Prestressed concrete bridge at the Kiruna mine 50
  E.1 Finite element model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
  E.2 Measurements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
    E.2.1 Measurements before breaking the bridge . . . . . . . . . . . . . . . . . . . 51
  E.3 Modal analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
  E.4 Preprocessing for increased SNR . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
    E.4.1 Anti-drifting preprocessing . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

F Concrete nine storey building — Luleå firehose tower 68
  F.1 Modal analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
  F.2 FE models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

G Kirchhoff plate — convex optimization without FE software interaction 81
  G.1 How to run the software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
  G.2 Preliminary results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

H Accelerometer calibration 94

I Implemented FEM updating methods 96
  I.1 Parametrization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
    I.1.1 Different choices of residual . . . . . . . . . . . . . . . . . . . . . . . . . . 97
    I.1.2 The objective function and its Taylor approximation . . . . . . . . . . . . . 99
  I.2 Derivatives of eigenvalues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
  I.3 Regularization with interpolating basis functions . . . . . . . . . . . . . . . . . 105
  I.4 Convex formulation of the optimization problem . . . . . . . . . . . . . . . . . . 110
    I.4.1 Sparse $l^1$-norm regularization . . . . . . . . . . . . . . . . . . . . . . . . 112
    I.4.2 Sparse regularization with $l^1$-norm and dictionaries . . . . . . . . . . . 113

J The source code 117
  J.1 Data storage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
    J.1.1 Shell, beam and spring elements . . . . . . . . . . . . . . . . . . . . . . . . 117
  J.2 AbaqusPkg package . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118
J.6.4 The FemOptParamConvMethod class ........................................... 197
J.6.5 The IBFemOptParamConv class .................................................. 198
  J.6.5.1 The method IBFemOptParamConv ....................................... 198
  J.6.5.2 The method convertFem2OptCon ....................................... 198
  J.6.5.3 The method convertFem2Opt ............................................. 198
  J.6.5.4 The method convertOpt2Fem ............................................. 198
  J.6.5.5 The method interpOptFunctions ....................................... 199
  J.6.5.6 The method sortRectMeshPts ......................................... 199
  J.6.5.7 The method zCoordsForPlane ......................................... 199
J.6.6 The NTOptimizer class .......................................................... 206
J.6.7 The ObjectiveFunction class .................................................. 208
  J.6.7.1 The method evaluate ...................................................... 208
J.6.8 The Optimizer class ............................................................. 210
J.6.9 The OptProblem class ............................................................ 210
J.6.10 The OutFunClass class .......................................................... 210
J.7 SimulatorPkg package .............................................................. 213
  J.7.1 The ResultType class ......................................................... 213
  J.7.2 The Simulator class ........................................................... 214
J.8 The UtilityPkg package .............................................................. 214
  J.8.1 The dlnode class .................................................................. 214
  J.8.2 The GeneralUtilities class ................................................... 216
    J.8.2.1 The method ??? for combining uniplanar patches with common
          edges ................................................................. 237
      J.8.2.1.1 signedArea ..................................................... 237
      J.8.2.1.2 lineSegmentsIntersect .................................... 238
      J.8.2.1.3 polyg2Clockwise ........................................... 239
    J.8.2.2 The method makeIndexVectors .................................... 239
    J.8.2.3 The method ReadGroupDefFile ..................................... 241
  J.8.3 The Geometry class ............................................................. 241
K  Paper A: Sensitivity-Based Model Updating for Structural Damage ... 243
L  Paper B: Modelling of Damage and its use in Assessment of a ... 268
Index ...................................................................................... 289
A Introduction to the appendices

This report is a continuation from research projects initiated in a previous SBUF project [Gri14]. The main results are presented in the Swedish part of this text. The appendices presents a more detailed description of some projects. Some presented preliminary preliminary results will be refined for publication in journal papers and in a forthcoming Ph.D. thesis.

Two main goals with this project have been

- Vibration measurement and modal analysis of new structures.
- A complete rewriting and generalization of the FEM updating software in [Gri14] to an object oriented software that is easier to modify for different structures and analysis methods.

We present some results and examples of how to use the FEM updating software in appendices B–G. Appendix I describes some of the implemented analysis methods. Finally, parts of the source code is described in more detail in Appendix J.

A.1 Object-orientated design for more flexible software

For simpler modifications of the software for different structures, optimization methods and for optional interaction with different Finite element (FE) software, the software is structured using an object-oriented design pattern called strategy pattern, as described, for example, in [http://www.oodesign.com/strategy-pattern.html]. We mainly follow the MATLAB style guidelines described in [Joh14]. The source code is structured into different packages AbaqusPkg, ..., OptimPkg that are organized as shown in Figure A.1–A.2.

A thorough introduction to object oriented programming is out of the scope of this report, but in this subsection we describe roughly how the object oriented structure is used for switching between either solving the FE generalized eigenvalue problem (I.2) in MATLAB or by interaction with the FE software Abaqus. Similar extentions can be written for interaction with other FE software.

All code for interaction with Abaqus is accessed via the methods provided by the large AbaqusPkg package. The smaller MatlabSimPkg package contains methods for solving the same generalized eigenvalue problem directly in MATLAB. As described in the following appendices, each implemented structure has its own directory with a Main file that sets up the geometry of the structure as well as which combination of optimization methods to use and which FE eigenvalue problem solver to use.

The Main file then calls a particular optimizer class from the OptimPkg package for performing the optimization. This package does not know if simulations (=solving the FE generalized eigenvalue problem) are done in MATLAB, by Abaqus or by some other software. It only knows of the the so-called interface SimulatorPkg. However, among the preparations done in the Main file before calling the optimizer class, is to call an appropriate constructor that extends the methods in the Simulator interface to to use the corresponding methods in either the MatlabSimPkg package or the AbaqusPkg package.

The AbaqusPkg is larger, for example, because the interaction with Abaqus requires the reading, parsing, modifying and writing of different text files. Moreover, Abaqus adds larger roundoff errors to the numbers in these text files, and currently, the Nelson method described in Section I.2 (page 101) only works with the MatlabSimPkg package, since even for a simple structure such as the concrete plate in Appendix B, Abaqus provides matrices that gives problems
A.2 Organization of the files

The software is contained in a main directory \texttt{SHM}, which contains the following four subdirectories:

- **FEM**: Here the finite element models for the different analyzed structures are stored. One subdirectory for each structure.

- **FEMU**: Here the FEM updating software is stored in two subdirectories.
  - **Impl**: for file specific for the implementation on each of the analyzes structures. One subdirectory for each structure, and one directory called **Template** with template files with some instructions inside on how to use them for application to anew structure.
  - **Library**: The implementation of an object oriented FEM updating software with all the code that is the same for all structures. One subdirectory for each structure. The important files are **Main.m**, which is the file to run in MATLAB, and the file **InputParameters**, which contains the most important parameters that must be defined for a new structure.

- **MeaData**: contains the measurements made on the different structures. One subdirectory for each structure.

- **ModalAnalysis**: contains the the files used for modal analysis on the different structures. One subdirectory for each structure.

For running the FEM updating software you need a computer with Abaqus and MATLAB installed on it. Also, in MATLAB you need to add the **Library** directory to the Matlab path with a command like `path(path,'D:\Matlab\SHM\FEMU\Library')`. For FEMU on the simulated Kirchhoff plate, no Abaqus or other FEM software is needed, but the convex optimization part uses the CVX package, which is available for free from [http://cvxr.com/cvx/](http://cvxr.com/cvx/).

The following appendices describes the vibration measurements, modal analysis and applications of this software done in this project for:

- A concrete plate.
- A concrete railway arch bridge.
- A Steel truss bridge.
- A prestressed concrete bridge.
- A concrete firehose tower.
- A simulated Kirchhoff plate.

Of these, FEM updating is not yet implemented for the prestressed concrete bridge or for the tower.
Figure A.1: Main structure of the source code. UML class diagram, as described, for example, in [link to Wikipedia Class Diagram]. For readable text, the right-hand half of the diagram is in Figure A.2.
Figure A.2: Right-hand half of the UML class diagram in Figure A.1.
B Concrete slab

The measurements for the concrete plate were first described in the previous SBUF report [Gri14]. In this appendix we explain how to run the FEMU (FEM updating) software, illustrate graphically the optimization done in a simple special case with two updating groups, and show some preliminary results. A more detailed description of measurements and damage detection results follow in the journal paper [GST17], which is included in this report as Appendix K page 243.

B.1 Running the FEMU software

Files are organized as described on page 35. Run the Main.m file in the subdirectory Impl/Plate for FEM updating on the plate with default settings. This generates plots like those in Figure B.1. Comments in this and other files explain the different parameters that can be changed, For example, the following parameters can be changed:

Different damages Chosen with the parameter dirName in InputParameters.txt.

Defining groups of elements In the file PlateGroupDefFile.txt, the material names for all 65 groups of elements are listed in an arbitrary order, one on each line.

Empty lines gives the division into groups, that is, names that follow without empty lines between are grouped into a group of elements. For example, one blank line between each group name gives 65 groups.

See also the documentation given in the beginning of PlateGroupDefFile.txt.

B.2 Simple FEM updating example with two groups of elements

Every structure has some resonance frequencies or eigenfrequencies and corresponding mode shapes describing standing waves oscillating with the corresponding eigenfrequency, see Figure B.1. Such modal data can be used as a “fingerprint” describing the dynamical properties of the structure. For a more formal description of theory and implemented methods, see Appendix I starting at page 96. Here we begin with a simple demonstration of the optimization done in a simple case where the finite element (FE) model of plate is divided into two halves (groups of elements) with elasticity modulus $E_1$ in one half and $E_2$ in the other half (parameters $P_1$ and $P_2$ in Figure B.2) below.

FE software compute the resulting predicted mode shapes and eigenfrequencies for chosen $E_k$, we compute predicted modal data (shapes and frequencies) with the Abaqus FE modeling software. We would like to adjust the parameters $E_1$ and $E_2$ to values that minimize the difference between the predicted mmodal data and corresponding modal data of the real structure, which can be obtained from so-called modal analysis of vibration measurements of the structure. The difference between predicted and measured modal data can then be measured by defining a so-called objective function, for example defined as in (I.10) on page 99.

As explained in Appendix K we performed laboratory vibration measurements and modal analysis on a concrete plate with five different levels of damage:

Case 0: No damage.

Case 1: A 6.5 mm deep notch.

Case 2: A 13.5 mm deep notch.
Figure B.1: The FEMU software generate plots of mode shapes and values of updating parameters. Undeformed plate, as well as measured and predicted mode shapes are plotted in three different colours in the same figure.
Case 0: Undamaged Case 1: 6.5×5 mm cut

Case 2: 13.5×3 mm cut Case 3 = Case 2 + 6.5 kN load crack

Case 4 = Case 3 + deeper cracks Summary of FEMU results

<table>
<thead>
<tr>
<th>Case</th>
<th># iter</th>
<th>(P₁, P₂)_{final} [GPa]</th>
<th>f(P₁, P₂)_{final}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 0</td>
<td>18</td>
<td>(38.025, 29.143)</td>
<td>0.128273</td>
</tr>
<tr>
<td>Case 1</td>
<td>14</td>
<td>(37.600, 27.372)</td>
<td>0.0830604</td>
</tr>
<tr>
<td>Case 2</td>
<td>18</td>
<td>(37.426, 26.448)</td>
<td>0.132399</td>
</tr>
<tr>
<td>Case 3</td>
<td>17</td>
<td>(36.674, 18.736)</td>
<td>0.334815</td>
</tr>
<tr>
<td>Case 4</td>
<td>17</td>
<td>(36.313, 13.552)</td>
<td>0.491771</td>
</tr>
</tbody>
</table>

Figure B.2: Objective function $f(P₁, P₂)$ for two updating parameters $P₁, P₂$. Red stars show the parameter values in each iteration, starting from $(P₁, P₂) = (10, 40)$ and converging towards the true minimum (located near the green star), until the Newton trust-region algorithm reaches its stop criterion and ends after 14–18 iterations in the different cases.
B.3 Regularization with shape functions

Case 3 = Case 2 + cracks caused by a 6.5 kN linear load.

Case 4 = Case 3 + deeper cracks from a larger linear load

Figure B.2 shows plots of the resulting objective function for these five cases. For this plate, we have implemented a Newton trust region optimization method for finding the minimum. It might miss the global minimum if there are more than one local minimum. Luckily, all five plots seem to indicate convex objective function with the global minimum being the only local minimum. Such convexity is not always the case, however. For example, we describe in Subsection 3.4.2 of Appendix L (starting at page 268) one case where the experimental setup introduces different local minima. The table in Figure B.2 shows that the Newton trust-region (with the restriction $10 \leq P_n \leq 40$) finds the global minimum in 14–18 iterations.

Another case where convexity is useful is demonstrated in Appendix G, where a convex reformulation of the optimization problem is used. Here the set of possible updating parameter values is a convex set, which can be used for solving the optimization problem faster, which can be useful for future FEMU for some of the larger structures with detailed FE models in this report.

Another advantage with a convex parameter set is that it was easier to implement a certain sparse optimization method, which also has the advantage of reducing the solution set and allowing for a smaller number of measurement points measuring the mode shapes [FG17].

B.3 Regularization with shape functions

More indecisive damage detection can be expected if the number of updating parameters are too large for the structure and measurement signal quality at hand. There are different regularization techniques for reducing the number of updating parameters in such cases. We have implemented linear interpolation technique described in some more detail in Appendix J.6.5, page 198.

The underlying idea is to smoothen the choice of updating parameters by letting the FEMU algorithm at hand choose parameter values freely on a subgrid under the side constraint that the parameter values in the other grid points then are chosen by linear interpolation.

The interpolation is used by the parameter coarseMeshInd in the file InputParameters.m.

- The default value is coarseMeshInd = 1:nrOfGroups. With 65 groups chosen as described in Section B.1, this gives 65 free updating parameters, as shown in the leftmost column in Figure B.3.
- The elements for the plate are numbered as follows:

```
1 6 11 16 21 26 31 36 41 46 51 56 61
2 7 12 17 22 27 32 37 42 47 52 57 62
3 8 13 18 23 28 33 38 43 48 53 58 63
4 9 14 19 24 29 34 39 44 49 54 59 64
5 10 15 20 25 30 35 40 45 50 55 60 65
```

Thus the choice

```
coarseMeshInd = [1 3 5 6 8 10 11 13 15 16 18 20 21 23 25 26 28 30 ... 31 33 35 36 38 40 41 43 45 46 48 50 51 53 55 56 58 60 61 63 65]
```

gives that the free updating parameters are the ones coloured grey in the top plot in the second column from the left and the parameter values for the white groups are chosen by linear interpolation, which gives the stripes in the second column in Figure B.3.

- Similarly, the third column corresponds to

```
coarseMeshInd = [1 5 6 10 11 15 16 20 21 25 26 30 ... 31 35 36 40 41 45 46 50 51 55 56 60 61 65]
```
and the fourth column corresponds to coarseMeshInd = [1 3 5 11 13 15 21 23 25]

Figure B.3: FEM updating results for the concrete plate gave better results when only the first three bending mode shapes were used. A damage is here indicated with blue, corresponding to a lower elasticity modulus value.

For a more clear indication of smaller damages (Case 1 and Case 2), we suspect that one should also update the torsional shear modulus when using torsion ode shapes, as done in [1DR03].
B.4 Regularized damage detection example

31 33 35 41 43 45 51 53 55 61 63 65].

- The rightmost column, finally, show the case with coarseMeshInd = 1:nrOfGroups and 13 vertical stripe groups chosen as described in Section B.1, which gives results similar to those in the third column in Figure B.3.

Note, however that linear interpolation is used on the parameters in the vector \( a \) of (I.5) (page 97), which in practice means that for the smooth results in Figure B.3 the parameter P0 in InputParameters.m was chosen to be a vector with all elements equal. With random elements in P0, this randomness be inherited by the results after FEM updating, as described in Remark 6, page 109.

B.4 Regularized damage detection example

Figure B.3 shows an overview of the FEM updating results with 65 or 13 groups defined as described in Section B.1. For the case with 65 groups, four different regularizations of the result are compared. The optimization algorithm is then free to chose arbitrary values for the elastic modulus within a preset range (for example 5 – 55 GPa) for the grey groups of elements in the top line of Figure B.3. Then the elastic modulus in other groups are chosen by linear interpolation from its neighbours (see Appendix J.6.5 for details). For example, if the forces and bending acting on the plate are such that the main expected damages are vertical cracks in Figure B.3 then the 13 groups on the right can be a suitable choice, or alternatively, the interpolation from 26 or 39 updating parameters, for a model that allow cracks with varying depth. Without advance knowledge of the most probable direction of cracks, 65 updating parameters can be a better result, but it can also give a more “noisy” result, which makes it more difficult to find small damages. In this case, the 21 updating parameter scenario could be a compromise for allowing different direction and varying depth of cracks, but still get some smoothing.

In row 2–6 of Figure B.3, all mode shapes from Figure B.1 are used for the FEM updating (see (I.10), page 99). For the real plate, a damage changes the stiffness \( EI \) by changing the cross section area and thus changing \( I \). In the FEM updating algorithm, the cross section areas are kept constant and the stiffness \( EI \) can instead be adjusted by changing the elastic modulus \( E \). For the two biggest damages, decreased elasticity modulus indicate damages at approximately the right place at least in the cases with 13 and 26 updating parameters, but a more precise indication of the size and location of the damage would be desirable.

In line 7–10, we have therefore instead compared with the elasticity modulus in previous measurements, by plotting the relative change \( \frac{E_n(x,y) - E_0(x,y)}{E_0(x,y)} \) of the elastic modulus \( E_n(x,y) \) in Case \( n \). This gives similar damage identification results in most cases, but more clear indication, for example, in Case 2 with 13 and 26 updating parameters. (In the following appendices, we follow common practice and plot the so-called damage index \( \frac{E_0(x,y) - E_n(x,y)}{E_0(x,y)} \), which is the same relative error with opposite sign.)

We get considerably more precise damage identification in line 11–19 of Figure B.3 where the same analysis is done as in row 2–10, but now only the three first bending mode shapes are used in the FEM updating, that is mode number 1, 4 och 5 in Figure B.1. For more sensitive and precise damage identification of smaller damages, we think that we also should update the torsional shear modulus in the horizontal direction of Figure B.3. This could give updating parameters that are more sensitive for changes in torsional modes (see [TDR05]).

In Figure B.3, we would ideally want relative change 0 of the elastic modulus in the undamaged parts of the plate, but instead we see an oscillation between positive and negative values
to the right and left of the actual damage. In the paper in Appendix \[\text{K}\], we investigate if some other kinds of regularization can reduce such oscillation effects and give more precise damage identification. Moreover, there the interpolation is done with the triangular element shape functions shown in Figure 3 of in Appendix \[\text{K}\] which reduces some unnecessary oscillations in Figure \[\text{B.3}\] that was caused by using rectangular element shape functions, following suggestions in \[\text{SD}^*\text{09}\] (see also Section \[\text{I.3}\]).

Note. The FEM updating software generates plots like those in Figure \[\text{B.1}\] and also saves the results in a text file. Figure \[\text{B.3}\] was obtained from a separate MATLAB-script that used data from those output files to produce the plots.
Concrete railway arch bridge(Långforsen, Kalix river)

The concrete railway arch bridge over Långforsen outside Kalix is a 177 meter long and 60 meter high bridge built 1960 (see Figure C.1). When the bridge’s owner Trafikverket wanted to increase the allowed speed and to increase the maximum allowed axis load from 225 to 300 kN, vibration measurements were done 2011, as well as measurements of strain and displacements with trains passing at different speeds. Measurements and models are also described in [SGT+17, WWZ+16].

Design and construction

The railway bridge over Kalix River at Långforsen has a total length of 177.3 m with a central arch of 89.5 m and two side spans of 42 m. The free spans have lengths of 13.0 + 12.8 + 12.6 + 87.92 + 12.6 + 12.8 + 13.0 m = 164.7 m. The bridge was designed by and built in 1960. The bridge consists of an arch which carries a reinforced concrete slab via underlying longitudinal and transversal concrete beams, connected trough fixed columns. The arch consists of a reinforced concrete hollow box girder with two hollow spaces. The cross section is lowest at the crown of the arc and highest at the connection to the arch abutment. The original train load corresponds to an axle load of 250 kN for the locomotive and a distributed load of 72 kN/m.

Geometry and material properties

The bridge was built with concrete Btg K 400 with present nominal characteristic compressive and tensile strengths of $f_{c,k} = 30.775$ MPa and $f_{t,k} = 1.95$ MPa and with a modulus of elasticity of $E_c = 32$ GPa. The foundations were cast with a slightly lower concrete quality, Btg K 300. The reinforcement was ribbed bars Ks 40 and high strength steel Ss70 with yield strengths of $f_{y,k} = 390$ and 720 MPa respectively. In 2009 six cylinders with a diameter of 95 mm were drilled out of the lower part of the arch. Three were tested in compression giving compressive strengths of 76.7; 79.8 och 65.2 MPa with a mean value of 73.9 MPa; three were splitted with splitting strengths of 1.85; 4.17 and 3.77 MPa. This corresponds to quality K80 according to

Figure C.1: The bridge over Kalix River at Långforsen.
BBK 94 (1994) with a characteristic compressive strength of $f_{ck} = 56.5$ MPa and a modulus of elasticity $E_{ck} = 38.5$ GPa.

C.1 Measurements

It was not clear if there was enough wind for excitation of ambient vibrations strong enough for modal analysis. Therefore, a T43 ra 240 railway engine was driven over the bridge at speeds between 35 and 63 km/h before each measurement. The engine weight is 72 tonnes (dynamic weight 79 tonnes), distributed on 8 wheels. The measurements began after the engine passing the bridge to have the same linear system (bridge only) as in our FE models and to exclude nonlinear effects, caused, for instance by noises from the wheels clattering against the rails and bridge endpoints clattering against the foundation.

There was a striking decrease of wind during the measurement days, which gradually decreased the signal-to-noise ratio (SNR). The SNR was, however, still good enough for clearly better correspondence between modeled and measured modal data (mode shapes and frequen-

![Setup 1](image1)

![Setup 2](image2)

![Setup 3](image3)

![Setup 4](image4)

![Setup 5](image5)

![Setup 6](image6)

![Setup 7](image7)

Figure C.2: Measurements on the Långforsen bridge 2011. The drawing shows the placement of the accelerometers. Photos show how the accelerometers were firmly attached to the deck and to the arch, as well as the train engine that was used for excitation.
C.2 Finite element models and preliminary FEMU results

A lot of work has been invested in different FEM models over the years. During 2011 two types of bridge models were developed with Abaqus/Brigade: A comprehensive model with foundations (Type I) and a simplified beam element model where the foundations have been exchanged to springs. (Type II). The advantage of the former model is that it is closer to the real bridge structure, and the predicted results from it should/could be more reliable and closer to the “real results”, but the disadvantage is that the problem size is rather high. The number of elements decreases from 93,910 in type I to 47,438 in type II and the number of variables decreases from 438,800 to 282,808. After further refinements we had

- A detailed shell element model with foundations.
- A simplified shell element model with foundations simulated with springs.
- A 3D Beam element model.
- A 3D planar beam element model.

Figure 4.5 in the main part of this report show some of the gradually improved FE models so far.

The Main files in the FEMU/Impl/LangforsenBridge directory computes FEM updating for the bridge with default settings. It produces plots like the ones in Figure C.3. It can update Table C.1: Summary of the measurements done with red color for the measurements that was used for the modal analysis.

<table>
<thead>
<tr>
<th>Name</th>
<th>Minutes</th>
<th>Wind [m/s]</th>
<th>f_samel</th>
<th>f_satt</th>
<th>v_train [m/s]</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1M0a</td>
<td>20</td>
<td>7.8–9.1</td>
<td>1200</td>
<td>7</td>
<td></td>
<td>Acc 6 not attached to bridge (missing screw thread)</td>
</tr>
<tr>
<td>S1M0b</td>
<td>40</td>
<td>5–8.6</td>
<td>1200</td>
<td>7</td>
<td></td>
<td>Acc 6 not attached to bridge (missing screw thread), damaged file</td>
</tr>
<tr>
<td>S1M2a</td>
<td>5</td>
<td>—</td>
<td>1200</td>
<td>5</td>
<td></td>
<td>Accidentally interrupted after 5 minutes.</td>
</tr>
<tr>
<td>S1M2b</td>
<td>7</td>
<td>1–4</td>
<td>1200</td>
<td>4</td>
<td></td>
<td>S1M2a continued for more minutes</td>
</tr>
<tr>
<td>S1M3</td>
<td>25</td>
<td>—</td>
<td>1200</td>
<td>100</td>
<td>55.35</td>
<td>49 km/h towards the hut (after a passage without measuring)</td>
</tr>
<tr>
<td>S2M1</td>
<td>25</td>
<td>—</td>
<td>1200</td>
<td>5</td>
<td>54.19</td>
<td>49 km/h</td>
</tr>
<tr>
<td>S2M1b</td>
<td>25</td>
<td>—</td>
<td>1200</td>
<td>5</td>
<td></td>
<td>Bonus: Light el/tele train going towards our hut.</td>
</tr>
<tr>
<td>S2M2</td>
<td>20</td>
<td>2–3</td>
<td>1200</td>
<td>100</td>
<td>35.64</td>
<td>49 km/h away from the hut. 100 Hz lowpass filter.</td>
</tr>
<tr>
<td>S3M1</td>
<td>25</td>
<td>—</td>
<td>1200</td>
<td>5</td>
<td>35.34</td>
<td>49 km/h away from the hut. 100 Hz lowpass filter.</td>
</tr>
<tr>
<td>S3M2</td>
<td>25</td>
<td>2–3</td>
<td>1200</td>
<td>100</td>
<td>19.54</td>
<td>20 km/h towards the hut</td>
</tr>
<tr>
<td>S4M1a</td>
<td>20</td>
<td>2–3</td>
<td>1200</td>
<td>100</td>
<td>28.94</td>
<td>± 30 km/h</td>
</tr>
<tr>
<td>S4M1b</td>
<td>25</td>
<td>2–3</td>
<td>1200</td>
<td>100</td>
<td>58.53</td>
<td>± 49 km/h away from the hut</td>
</tr>
<tr>
<td>S5M1a</td>
<td>25</td>
<td>2–3</td>
<td>1200</td>
<td>100</td>
<td>40.11</td>
<td>± 40 km/h away from the hut</td>
</tr>
<tr>
<td>S5M1b</td>
<td>25</td>
<td>2–3</td>
<td>1200</td>
<td>100</td>
<td>28.75</td>
<td>Towards the hut, p-plade pointing towards west for accelerometer 2</td>
</tr>
<tr>
<td>S5M3</td>
<td>27</td>
<td>—</td>
<td>400</td>
<td>100</td>
<td>15.09</td>
<td>± 10 km/h away from the hut</td>
</tr>
<tr>
<td>S6M1</td>
<td>42</td>
<td>—</td>
<td>400</td>
<td>100</td>
<td>0.33</td>
<td>± 2 km/h away from the hut</td>
</tr>
<tr>
<td>S6M2</td>
<td>27</td>
<td>—</td>
<td>400</td>
<td>100</td>
<td>15.09</td>
<td>± 10 km/h away from the hut</td>
</tr>
<tr>
<td>S7M1</td>
<td>27</td>
<td>—</td>
<td>400</td>
<td>100</td>
<td>15.09</td>
<td>± 10 km/h away from the hut</td>
</tr>
<tr>
<td>S7M2</td>
<td>35</td>
<td>—</td>
<td>400</td>
<td>100</td>
<td>0.33</td>
<td>± 2 km/h away from the hut</td>
</tr>
</tbody>
</table>

Figure C.2 shows the accelerometer mounting, the measurement setups and the engine. Table H.1 summarizes some data for the measurements done.

Modal analysis was performed in the software ARTeMIS 4.0 using the principal components stochastic subspace Identification (SSI) method.
both boundary conditions (modelled as springs) an Elasticity modulus in different parts of the bridge.

The main problems were that the upper right bending mode shape in Figure C.3 has mode frequency only has mode frequency 2.78 Hz, and FEM updating did not bring it closer to the measured mode frequency 3.06 Hz. One possible reason for this is that the beam element model was developed in order to decrease its size and therefore to be more computation efficient both for dynamic response simulations and for finite element model updating, but it also suffered from the drawback that the arch can not undergo torsion.

We have therefore continued refining the FE model, both for the arc, but currently also by a recently added 2D track model that finally changed the mode frequency to 3.55 Hz. However, similar to the concrete plate, FEMU moves frequencies for torsion modes more far away from the corresponding measured frequencies. Therefore, we think that one further necessary improvement is to adapt the FEMU software to updating shear modulus for some degrees of freedom, as suggested in [TDR03].

Figure C.3: FEM updating results for the bridge over Långforsen river.
D  Steel truss bridge over Åby river

The measurements and the modal analysis results are described, for example, in the conference paper [BHN+14], the SBUF report [Gri14], in the Swedish main part of this report and some papers cited there.

FEM updating

Run the Main.m file in the directory FEMU/Impl/AbyBridge for running FEMU on the bridge over Åby River with default settings. It produces plots like those in Figure D.1.

In the measurements on a damaged bridge, some of the beams have been loaded enough for clearly visible plastic deformation, which, however changes neither the elastic modulus or the cross section areas, so we do not expect the visible damages to change the dynamic properties of the bridge. With FEM updating, we do instead hope to detect damages in the connections between the beams that are not visible for the eye.

However, we think that updating only the elasticity modulus is not enough for good results when using both bending and torsion modes in the FEM updating (as in Section B.4, where we got best results by using only bending modes). Therefore we are currently investigating how to adapt the FEMU software to also update the shear modulus in different parts of the analyzed structure.

Further results will be presented in future papers and thesis.

Figure D.1: FEM updating results for the bridge over Åby River.
E  Prestressed concrete bridge at the Kiruna mine

Several experiments have been conducted by Luleå University of Technology and partners on a 121.5 m long five-span continuous prestressed concrete bridge in Kiruna, Sweden. See, for instance, the project report [EBN+15]. For some more recent result and further description of the FE models used, see the paper [HGS+16] included as Appendix refapp:IABSEPaperKiruna16.

The Kiruna Bridge was built in 1959, connecting the city center to the mine. The bridge was post-tensioned in two stages during the construction in 1959, starting with six cables of the central segment, followed by four and six cables of the western and eastern segments, respectively. Due to the extensive ground deformation and settlement caused by the underground mining operation, the bridge was closed in 2013 and then demolished in September 2014. One experiment that was performed before demolition was to load the bridge to failure. Before and after doing this, different measurements were done on the bridge. In this appendix, the main focus is on ambient vibration measurements performed before and after loading the bridge to failure, as well on the modal analysis performed to obtain the modal data (mode shapes, eigenfrequencies and damping ratios) that are summarized in Table E.1 and figures E.9–E.12 below.

The mode shapes and eigenfrequencies can be used for damage identification as described in other sections of this report. Such methods are important for maintenance of different structures, for extending their life span and for better knowledge of their load carrying capacity. For the large and detailed model needed for this bridge, the sparse regularized FEM updating (FEMU) methods described in Section L and Appendix G and L may be useful for faster and more precise FEMU (than the previously implemented Newton trust-region method) for the large and detailed FE model needed for this bridge.

E.1 Finite element model

Two 3D models have been built in the FE modeling software Abaqus, one using shell elements and one using a combination of shell and beam elements.

Predictions obtained from these two models are well consistent with mode shapes and eigenfrequencies computed from acceleration measurements on the bridge before and after loading it to failure, see Figure E.9, E.12 and the paper [HGS+16], included in this report as Appendix L.

E.2 Measurements

Accelerometer measurements of ambient vibrations were performed in May 2014 on the functional bridge and then again during two weekends in August after loading the bridge to failure. The weather conditions during the measurements are summarized in Figure E.1.

Measurements were done with six Colibrys SF3000L triaxial accelerometers (two with malfunctioning measurements in one horizontal direction) connected with 40–60 m long six wire twisted pair cables to an MGC-Plus data acquisition system using AP801 cards with sample rate 800 Hz. The accelerometers were firmly attached to the bridge with expansion bolts and adjusted to the horizontal plane with three screws, see Figure E.2. The triaxial accelerometers were calibrated as described in [FGS13]. Figure E.3 shows the distribution of the accelerometers on the bridge.
E.2 Measurements

E.2.1 Measurements before breaking the bridge

The ambient vibrations caused by the wind were measured May 17-18. The wind in Kiruna then was in the range 1–12 m/s. The average wind speed was just over 6 m/s, mainly in a direction perpendicular to the bridge deck, as shown in Figure E.1. This was good conditions for measuring ambient vibrations, but unfortunately the electrical power supply at the bridge was malfunctioning, causing small sparks when connecting or disconnecting some of the twisted pair cables and clearly visible interferences in some of the corresponding measurements, see Figure E.4. However, we only had one weekend allowed for measurements, so we had to use provided electric power. Of the 38 measurement points shown in Figure E.3 measurements C4, E4 and almost all reference measurements with accelerometer R6 contained interferences of the kind shown in Figure E.4. Red vertical lines shows identified sharp jumps of the standard deviation. The power spectrum still look somewhat similar to the power spectra from other

Figure E.1: The Kiruna temperature, wind and air pressure during the measurements in May 17–18 (1), August 16–17 (2) and August 21–23 (3). The measurement days are marked with light blue shading. Statistics from [http://rl.se](http://rl.se)
measurements, so a simple attempt to “correct” the measurement errors could be to multiply with different constants in different intervals. Then the big question would be which constants to multiply with, since the wrong ones gives wrong amplitude in the corresponding measurement points for the mode shapes computed below. For the reference accelerometer, the consequences

Figure E.2: Horizontal attachment of accelerometers.

Figure E.3: Distribution of measurement points on the bridge. The distance between measurement points in meters were measured on the bridge, except for the distance between E3 and C2, that was estimated from this drawing. Measurements from the points C4, E4 and R6 are excluded from the modal analysis because of interferences most likely caused by malfunctioning electric power (see Figure E.5).
Figure E.4: The top row shows sharp jumps of the standard deviation in the measurements C6. The preprocessing described in Appendix F.1 removes nonlinear drifting. This gives the acceleration measurements in the third row. The last row shows the corresponding power spectra. Vertical lines in the power spectra indicate the local maxima closest to the mode frequencies in figures E.6–E.8.
Figure E.5: The top rows show the measurements I4 and R6, that should be nearly identical but the R6 measurements have severe interferences primarily in the x-direction measurements. Row 3–4 shows the corresponding power spectra, which show similarities. The sudden jumps between different standard deviations in the I6X measurements primarily change the high frequency contents and changes the heights of the peaks. The latter will cause incorrect amplitudes and deformed mode shapes if measurements with such disturbances are used for modal analysis.
are even worse, since wrong constants would give incorrect mode shape amplitudes for all measurement points, with different errors for the nine different measurement setups.

Moreover, we put measurement points I4 and R6 very close (see Figure E.3), so that they should give nearly identical measurements, but Figure E.5 shows that even after multiplication with the right constants in the right intervals, the simultaneous measurements I4 and R6 would look quite different. Although their power spectra looks similar for the x-direction, the height of the peaks will be wrong and give incorrect mode shape amplitudes.

Therefore we decided to consider all measurements from the points C4, E4 and R6 to be unreliable and have excluded them from all the analysis described below.

Moreover, measurement points I2 and H2 coincide, and H2 seems to have higher power spectrum signal-to-noise ratio, so I2 is also excluded from the modal analysis.

**Measurements after breaking the bridge**

After loading the bridge to failure, new measurements were done August 16–17. Now with electrical power from a petrol-driven generator, but unfortunately with much less wind. New longer measurements were therefore done August 21–23, still with rather light wind (see Figure E.1). The wind was now also most of the time in the direction of the bridge deck during both weekends. Therefore, we were not sure if there were enough ambient vibrations for modal analysis.

Measurement setups were the same as for the previous measurement (see Figure E.3).

**E.3 Modal analysis**

As described above, we have measurements from three weekends:

**Kiruna 14b** Measurements August 16–17 after loading the bridge to failure.
**Kiruna 14c** Measurements August 21–23.

**Modal analysis before breaking the bridge**

Figures E.6–E.8 show the modal analysis results for the undamaged bridge. Mode shapes, frequencies and damping were computed with seven different modal analysis methods with the software ARTeMIS.

**Modal analysis after breaking the bridge**

Due to the minimal excitation from wind, as well as from and cars passing on a road under the bridge, we expected a low signal-to-noise ratio (SNR) in the measurements. One way to increase the SNR is to make longer measurements, so we made the 14c measurements longer than the 14b measurements and also used a combination of the 14b and 14c measurements for further increasing the SNR. However, for measurement setups F, G, H, I, we use only the Kiruna 14b measurements because of the following errors in the 14c measurements:

- F3X, F3Y, F5X and F5Y missing
- G1Y and G3X missing. G3Y looks a bit strange.
- Higher noise floor for the H3Z power spectrum and this higher noise floor remains if the measurements 14b and 14c are combined, as described in Section E.4 below.
- I2Z missing.
Figure E.6: FDD modal analysis results for the undamaged Kiruna bridge. Since Measurement point E4 is excluded from the analysis, and since E2, B2 and F2 contain no z-direction measurements (see Section F) the z-direction mode shape amplitude in E4 is computed by extrapolation from the values in E3 and E5 (see Figure E.3). Thus the high amplitude in E4 in mode 14 should be taken with a grain of salt.

In figures E.6–E.8, the LKAB mine is always to the left and the city centre to the right.
EFDD: Similar mode shapes and frequencies

<table>
<thead>
<tr>
<th>Frequency [Hz]</th>
<th>Damping [%]</th>
<th>Complexity [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: 1.914</td>
<td>1.209</td>
<td>0.322</td>
</tr>
<tr>
<td>2: 4.532</td>
<td>0.856</td>
<td>0.28</td>
</tr>
<tr>
<td>4: 4.904</td>
<td>0.682</td>
<td>2.000</td>
</tr>
<tr>
<td>8: 5.916</td>
<td>0.779</td>
<td>4.575</td>
</tr>
<tr>
<td>9: 7.129</td>
<td></td>
<td>3.194</td>
</tr>
<tr>
<td>10: 8.357</td>
<td>1.195</td>
<td>1.345</td>
</tr>
<tr>
<td>12: 14.055</td>
<td>0.816</td>
<td>22.696</td>
</tr>
<tr>
<td>16: 19.582</td>
<td>0.185</td>
<td>66.601</td>
</tr>
<tr>
<td>19: 19.617</td>
<td>0.327</td>
<td>13.957</td>
</tr>
</tbody>
</table>

Additional modes found when searching in the 0-25 Hz frequency band.

CFDD: No new mode shapes, similar frequencies

<table>
<thead>
<tr>
<th>Frequency [Hz]</th>
<th>Damping [%]</th>
<th>Complexity [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: 1.914</td>
<td>1.148</td>
<td>0.322</td>
</tr>
<tr>
<td>2: 4.53</td>
<td>0.538</td>
<td>0.28</td>
</tr>
<tr>
<td>4: 4.902</td>
<td>0.463</td>
<td>2.000</td>
</tr>
<tr>
<td>9: 5.918</td>
<td>0.484</td>
<td>4.575</td>
</tr>
<tr>
<td>12: 8.353</td>
<td>0.765</td>
<td>1.345</td>
</tr>
<tr>
<td>16: 14.055</td>
<td>0.508</td>
<td>22.696</td>
</tr>
</tbody>
</table>

Additional modes found when searching in the 0-12.5 Hz frequency band.

SSI-UPC: More realistic shape for two modes

<table>
<thead>
<tr>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping [%]</th>
<th>Std. Damping [%]</th>
<th>Complexity [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: 1.911</td>
<td>0.007</td>
<td>1.231</td>
<td>0.291</td>
<td>0.119</td>
</tr>
<tr>
<td>16: 13.447</td>
<td>0.059</td>
<td>1.682</td>
<td>0.549</td>
<td>3.878</td>
</tr>
<tr>
<td>19: 20.011</td>
<td>0.069</td>
<td>1.293</td>
<td>0.445</td>
<td>56.360</td>
</tr>
</tbody>
</table>

Additional modes found when searching in the 0-25 Hz frequency band.

Figure E.7: EFDD, CFDD and SSI-UPC modal analysis results for the undamaged Kiruna bridge.
SSI-PC: No modes found
SSI-CVA: One new mode, others like SSI-UPC
SSI-UPC Merged: six new or differently shaped nodes

---

**Figure E.8**: SSI-PC, SSI-CVA and SSI-UPC Merged modal analysis results for the undamaged Kiruna bridge.
Table E.1: Overview of ARTeMIS modal analysis results for measurements on the functional bridge with notation explained on page 59. Parentheses in the table indicate mode shapes that are more noisy than the others. Mode shapes 14 and 15 might be the same. For modes marked with boldface, modal data are listed in Table E.2 and plotted in figures E.9–E.12.

<table>
<thead>
<tr>
<th>Mode nr</th>
<th>FDD</th>
<th>EPDD</th>
<th>CFDD</th>
<th>SSI-UPC</th>
<th>SSI-PC</th>
<th>SSI-CVA</th>
<th>SSI-UPC Merged</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a, b/c</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>a</td>
<td>b, (b/c), bc</td>
<td>a, b/c, bc</td>
<td>a, b/c, bc</td>
<td>a</td>
<td>a, a, b/c, bc</td>
<td></td>
</tr>
</tbody>
</table>

Table E.2: Comparison of mode frequencies $f$ and damping $\xi$ on the damaged bridge against the same modal data on the damaged bridge and the frequencies predicted by the FE models. We use the notation “value ± standard deviation”. $\Delta f$ denotes the relative change of the mode frequencies after loading the bridge to failure.

For all other measurements, the power spectra generally looks similar in the 14b and 14c measurements, but with slightly lower and smoother noise floor for the longer 14c measurements.

As indicated in Figure E.3, measurement points D3 and D4 coincide with I2 and I4, respectively. However, measurement I2Z is missing and D4 has higher noise floor than I4. Therefore, we use the measurements D3 and I4 for these points.

Altogether, this gave three different combinations measurements available for analysis:

- **a**: Measurements on the undamaged bridge.
- **b**: August 16–18 measurements on the damaged bridge.
- **b/c**: August 16–18 measurements for measurement setups F, G, H, I on the damaged bridge.
- **bc**: August 21–24 measurements for all other setups.

Combination of the August 16–18 and August 21–24 measurements for all other setups.

Modal analysis was performed with all methods available in the software ARTeMIS. This resulted in 25 pages of modal analysis results that are available from the authors, but in this report we
prefer to give an overview in Table E.1 of the vibration modes found for each analysis method and combination of measurement data. The vibration modes found in measurements 14a are there numbered from 1 to 19. One additional mode found only in the measurements 14b and 14c is labeled 10b.

Note, for example, that modes number 10 and 16 were found for the damaged bridge only when using the combined measurements b/c.

For all modes found both before and after loading the bridge to failure, Table E.2 compares the found mode frequencies on the damaged and undamaged bridge. For mode 16, some predicted modes were close in frequency but none with matching mode shapes. The measured frequencies are lower for the damaged bridge, which also is what to expect from theory.

Remark 1. It is well-known that it is more difficult to compute the correct damping than the correct mode frequency. For \( f_{\text{damaged}} \) and \( \xi_{\text{damaged}} \) found for different methods and signals in Table E.1, we chose the ones with best looking mode shapes and smallest frequency standard deviation for Table E.2. Frequencies are quite similar for other choices, but the “value \( \pm \) standard deviation” notation should not be interpreted literally as confidence interval with high probability containing the correct value of the computed parameter. For example,

- for mode 4 alternatively \( f_{\text{damaged}} = 4.914 \pm 0.000093 \), \( \xi_{\text{damaged}} = 0.5843 \pm 0.000328 \) for SSI-UPC merged analysis of the b measurements.
- For mode 7 alternatively \( f_{\text{damaged}} = 5.366 \pm 0.001977 \), \( \xi_{\text{damaged}} = 3.995 \pm 0.1008 \) for SSI-UPC merged analysis of the bc measurements.
- For mode 9 alternatively \( f_{\text{damaged}} = 5.722 \pm 0.000684 \), \( \xi_{\text{damaged}} = 0.7255 \pm 0.004767 \) for SSI-UPC merged analysis of the bc measurements.

Figures E.9–E.12 shows a comparison of the computed and predicted mode shapes. It also compares with vibration modes computed from measurements performed in July 2008 and in the turn of the month January–February 2008. The 2008 summer and winter measurements are described in more detail in [ESEE11]. We do however consider the modal analysis results from the 2014 measurements to be more reliable, since the accelerometers were not firmly attached to the bridge in the 2008 measurements.

Conclusions

- Table E.2 shows that mode frequencies are between 1.0 % and 11.4 % lower for the damaged bridge than for the undamaged bridge.
- In figures E.9–E.12 the predicted and measured modes are quite similar for vibration modes 1, 2, 10, 11 and 12, as well as number 13 if measurements on the rightmost part of the bridge (starting with I3, I4, I5 in Figure E.3) are ignored. Of these, number 11 and 13 was found only in the measurements on the functional bridge.

These are the vibration modes that seem most useful for damage identification. Mode 10 is also one of those that could be reliably detected for the damaged bridge only for the combined measurements bc.

Thus this combination was successful in providing better results, but an important question for future research is to investigate if such a combination can result in some bias or other side-effect on modal data computed with the SSI methods, such as the differences mentioned in Remark 1. When using these results for damage identification, without such an investigation, one possible precaution when using the results for damage detection is to avoid combining the 14b and 14c measurements if similar modal data and mode shape can be computed without such a combination.
### E.4 Preprocessing for increased SNR

<table>
<thead>
<tr>
<th>Shell element</th>
<th>Shell-beam</th>
<th>Undamaged bridge</th>
<th>Damaged bridge</th>
<th>July 2008</th>
<th>Jan-Feb 2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.8692 Hz</td>
<td>1.8851 Hz</td>
<td>1.851 Hz</td>
<td>1.91 Hz</td>
<td>1.8172 Hz</td>
<td>f = ±1.851 Hz</td>
</tr>
<tr>
<td>4.0998 Hz</td>
<td>4.437 Hz</td>
<td>4.437 Hz</td>
<td>4.21 Hz</td>
<td>4.2269 Hz</td>
<td>f = ±4.2269 Hz</td>
</tr>
<tr>
<td>4.6862 Hz</td>
<td>4.7456 Hz</td>
<td>4.7456 Hz</td>
<td>4.91 Hz</td>
<td>4.7311 Hz</td>
<td>f = ±4.7311 Hz</td>
</tr>
<tr>
<td>5.0725 Hz</td>
<td>5.032 Hz</td>
<td>5.032 Hz</td>
<td>5.0725 Hz</td>
<td>5.1202 Hz</td>
<td>f = ±5.1202 Hz</td>
</tr>
<tr>
<td>5.4743 Hz</td>
<td>5.4223 Hz</td>
<td>5.4223 Hz</td>
<td>5.428 Hz</td>
<td>5.443 Hz</td>
<td>f = ±5.443 Hz</td>
</tr>
<tr>
<td>5.388 Hz</td>
<td>5.543 Hz</td>
<td>5.543 Hz</td>
<td>5.289 Hz</td>
<td>5.388 Hz</td>
<td>f = ±5.388 Hz</td>
</tr>
</tbody>
</table>

Figure E.9: Comparison of analyzed and predicted mode shapes 1.
<table>
<thead>
<tr>
<th>Shell element</th>
<th>Shell-beam</th>
<th>Undamaged bridge</th>
<th>Damaged bridge</th>
<th>July 2008</th>
<th>Winter Jan-Feb 2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f = 5.7191$ Hz</td>
<td>$f = 5.7837$ Hz</td>
<td>5.907 Hz</td>
<td>9: 5.97 Hz</td>
<td>$f = 5.87 \pm 0.03$</td>
<td>$f = 6.39 \pm 0.03$</td>
</tr>
<tr>
<td>$f = 6.9174$ Hz</td>
<td>$f = 6.7435$ Hz</td>
<td>7.092 Hz</td>
<td>10: 6.882 Hz</td>
<td>$f = 6.83 \pm 0.01$</td>
<td>$f = 7.41 \pm 0.03$</td>
</tr>
<tr>
<td>$f = 7.3905$ Hz</td>
<td></td>
<td>7.499 Hz</td>
<td></td>
<td>$f = 7.37 \pm 0.03$</td>
<td>$f = 7.92 \pm 0.02$</td>
</tr>
<tr>
<td>$f = 8.2920$ Hz</td>
<td></td>
<td>8.182 Hz</td>
<td>12: 7.876 Hz</td>
<td>$f = 8.51 \pm 0.05$</td>
<td>$f = 9.42 \pm 0.05$</td>
</tr>
<tr>
<td>$f = 8.2454$ Hz</td>
<td></td>
<td>8.837 Hz</td>
<td>13: 8.976 Hz</td>
<td></td>
<td>$f = 10.50 \pm 0.03$</td>
</tr>
<tr>
<td>$f = 10.235$ Hz</td>
<td></td>
<td>9.851 Hz</td>
<td>14 (or 15?): 8.976 Hz</td>
<td>$f = 10.97 \pm 0.13$</td>
<td>$f = 12.56 \pm 0.24$</td>
</tr>
</tbody>
</table>

Figure E.10: Comparison of analyzed and predicted mode shapes 2.
Figure E.11: Comparison of analyzed and predicted mode shapes 3.

Figure E.12: Comparison of analyzed and predicted mode shapes 4.
E.4 Preprocessing for increased SNR

As described above, we did long measurements and also used combinations of long measurements to increase the low signal-to-noise ratio (SNR) that we expected because we had to measure ambient vibrations when there was almost no wind.

Combination of two signals adds some error to the computed modal data, but at least for the FDD methods, we expected this error to be quite small, since the cross-corellations computed by these methods should be computed on the Fourier transformed signals for efficiency, and the ARTeMIS documentation confirmed that the discrete Fourier transforms (DFT) are computed by cutting the signal into shorter overlapping parts, computing the DFT of each and then computing the average of (for our signal lengths and sampling rates) several hundred such Fourier transforms for noise reduction. In this average only one or at most two terms contain some error from the discontinuity occurring where the two measurement signals are joined, so therefore it was certainly possible to get decreased SNR from joining two long measurements, and therefore worth trying. For the SSI methods, it is less clear exactly how the discontinuities affect the computed modal data, but we have seen above that mode shape number 10 was one of those that best matched the predicted mode shapes, and for the damaged bridge, the mode shapes matched well only when applying the SSI-UPC merged analysis to the combined measurements.

![Preprocessing demo](image)

Figure E.13: Preprocessing demo. The lowermost row shows the power spectral density before (red) and after the anti-drifting preprocessing (blue).
Nonlinear drifting of voltages

The accelerometers were connected to the MGCplus data acquisition system with 40–60 m long 6 wire twisted pair cables (one pair of wires for each measurement direction). Since there are no wires for power, each accelerometer is powered separately by 24 1.5 V AA batteries. The twisted pair wires have shown to have heat dependent resistance in previous measurements, causing nonlinear drifting of the measured output voltages from the sensors used. This phenomenon has been observed in previous measurements both with accelerometers and with other sensors, and sunny days it has been possible to reduce this drifting by putting the cables in the shadows.

However, it was impossible to avoid significant nonlinear drifting for some of the long measurements at the Kiruna bridge, as can be seen in the top row plots of Figure E.13. Even more severe drifting was observed for the firehose tower in Appendix F due to even longer measurements, partially sunny days and greenhouse effect caused by the glass walls of the tower.

This drifting had to be reduced, because the computed Fourier transforms in the FDD method are very sensitive to the discontinuities in the periodized signal and in the point of transition between the two measurement signals in the top row of Figure E.13. The drifting is also expected to disturb the SSI methods, although in other ways than via computed DFTs.

E.4.1 Anti-drifting preprocessing

Only a bandpass filtering was not enough to reduce the drifting (it would only change the red power spectral densities in the lowermost row of Figure E.13 in the 0–0.05 Hz frequency band). Therefore, the preprocessing done for the Kiruna bridge and for the firehose tower was done in the following way. For the Kiruna bridge and two combined measurements, the following preprocessing was perform on both signals individually before joining them. The plots referred to in the following list are always the plots in Figure E.13.

1. First, the measured signals in the top row are translated from voltages to corresponding acceleration with the calibration technique described in [FGS13]. This gives the plots in the second row. This calibration technique assumes certain biases to be time-invariant, which is not the case for the drifting signal. Therefore, the discontinuities at the endpoints and in the transition between the two signals can even get bigger after calibration, but the error in bias is eliminated in the following steps, which removes this problem, so we chose to still do the calibration.

2. The drifting (red curve in the second row) is removed in three steps, here with $f_{\text{Detrend}} = 0.05$ Hz:

   (a) Make the signal $1/f_{\text{Detrend}}$ seconds longer by padding after the last sample with a smooth cosine transition from the average value in the last, say, 0.1 seconds to the average value in the first 0.1 seconds. (The underlying thought was to replace the discontinuity in the periodized signal that the DFT sees with something that will be removed by the highpass filtering.)

   (b) Highpass filter with a raised cosine window that increases from 0 at $f_{\text{Detrend}}/10$ to 1 at $f_{\text{Detrend}}$. This removed almost all drifting in the cases we investigated more closely, but just in case there is any noticeable remaining linear drifting we also do the following.

   (c) Make a least squares approximation with a straight line and subtract this line from the signal.
This gives the signals in the third row.

3. The plots in the lowermost row show the resulting power spectra before (red) and after this detrending preprocessing (blue).

(The plots in the forth row show time-frequency analysis with a continuous wavelet transform (see, e.g., [Gri02]), which were plotted as a precaution, but showed no interssting information or misbehaviour of the method.)

Just as in Figure E.13 the blue curves in Figure E.14 show the spectrum after the anti-drifting was applied, and the red curves show the same spectra without this anti-drifting. For some of the signals, this removed quite big clearly visible disturbances in the 0–3 Hz frequency band, so it is unlikely that mode shapes 1 and 2 in Figure E.9 could have been detected without this anti-drifting preprocessing. Figure E.4 also shows both the drifting and the corresponding changes in the spectrum for some of the shorter measurements on the undamaged bridge.
Measurement E3, E5, G3, H3 and I3 had more interferences giving higher noise floor than other measurements at higher frequencies.
Concrete nine storey building — Luleå firehose tower

As a complement to the concrete plate and the three different bridges in previous appendices, we have also made vibration measurements, modal analysis and FE modeling of a at the Luleå fire station. This is a completely different structure that for symmetry reasons, for example can have multiple modes with the same or almost the same mode frequency. This puts some more demands on the modal analysis software (covered by ARTeMIS) and on the FEM updating methods (covered by Nelson’s method, see Section 1.2 starting on page 101).

The tower has three concrete walls covered with bricks and a front wall consisting of large glass windows with steel beams between them. See Figure F.1. Seen from the front, the

Figure F.1: Main structure of the firehose tower.
rightmost brick wall contains one big “window” with corresponding three smaller windows in
the concrete wall on floors 3, 5 and 7, as shown in Figure F.1.

Inside the tower, there are grating stairs connecting 20 grating half-floors, numbered 0-19
in the measurement plan in Figure F.2. The more precise placement of each accelerometer is
given in Table F.1 for completeness.

Long measurements because of no wind

We only had the measurement equipment available for renting one week and that week there
was almost no wind, as shown in Figure F.3. Those measurements was at a weather station
 closer to the sea, so the actual wind at the tower was most likely lower. Therefore, most
measurements were several hours long.

F.1 Modal analysis

The long measurements, hot summer days and greenhouse effects from the glass wall gave severe
nonlinear drifting of the voltages, which were reduced by the same anti-drifting technique as
described in Section E.4

Figures F.4–F.10 show the modal analysis results. Accelerometers were placed in the corners
of the solid green triangles. Other triangles are obtained from interpolation or extrapolation.
Altogether 20 different modes. A few observations:

Table F.1: The position of the accelerometers, ordered as in Figure F.2. For each accelerometer,
the position is given as the horizontal distance $d$ from and vertical height $h$ above the inner-
upper corner of the steel beam at floor level below the window.

<table>
<thead>
<tr>
<th></th>
<th>$d$</th>
<th>$h$</th>
<th></th>
<th>$d$</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_3$</td>
<td>13.8</td>
<td>14</td>
<td>$B_3$</td>
<td>35.3</td>
<td>19.3</td>
</tr>
<tr>
<td>$R_1$</td>
<td>—</td>
<td>—</td>
<td>$B_4$</td>
<td>203</td>
<td>22</td>
</tr>
<tr>
<td>$C_5$</td>
<td>44</td>
<td>15</td>
<td>$C_4$</td>
<td>33</td>
<td>24.5</td>
</tr>
<tr>
<td>$A_2$</td>
<td>220.5</td>
<td>14</td>
<td>$R_6$</td>
<td>218</td>
<td>19</td>
</tr>
<tr>
<td>$A_5$</td>
<td>46</td>
<td>11.5</td>
<td>$B_2$</td>
<td>27</td>
<td>19</td>
</tr>
<tr>
<td>$C_2$</td>
<td>221.3</td>
<td>14</td>
<td>$B_5$</td>
<td>212</td>
<td>19</td>
</tr>
<tr>
<td>$C_5$</td>
<td>29</td>
<td>14.5</td>
<td>$D_5$</td>
<td>47</td>
<td>20</td>
</tr>
<tr>
<td>$D_3$</td>
<td>220.3</td>
<td>16.5</td>
<td>$D_2$</td>
<td>201.5</td>
<td>16.5</td>
</tr>
<tr>
<td>$D_4$</td>
<td>26</td>
<td>17.5</td>
<td>$E_3$</td>
<td>48</td>
<td>18.5</td>
</tr>
<tr>
<td>$E_2$</td>
<td>217.5</td>
<td>17</td>
<td>$E_4$</td>
<td>213</td>
<td>16</td>
</tr>
<tr>
<td>$E_3$</td>
<td>25</td>
<td>24.5</td>
<td>$F_3$</td>
<td>48</td>
<td>19.5</td>
</tr>
<tr>
<td>$F_3$</td>
<td>219.5</td>
<td>17</td>
<td>$F_2$</td>
<td>47.5</td>
<td>19.5</td>
</tr>
<tr>
<td>$F_4$</td>
<td>38</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$G_3$</td>
<td>231</td>
<td>-10</td>
<td>$G_2$</td>
<td>208</td>
<td>19.3</td>
</tr>
<tr>
<td>$G_4$</td>
<td>40</td>
<td>18</td>
<td>$G_3$</td>
<td>48</td>
<td>18.3</td>
</tr>
<tr>
<td>$H_2$</td>
<td>228</td>
<td>14</td>
<td>$H_3$</td>
<td>209.5</td>
<td>49</td>
</tr>
<tr>
<td>$H_3$</td>
<td>35.5</td>
<td>19.5</td>
<td>$H_4$</td>
<td>47</td>
<td>20</td>
</tr>
<tr>
<td>$I_2$</td>
<td>230</td>
<td>-13</td>
<td>$I_3$</td>
<td>205</td>
<td>22</td>
</tr>
<tr>
<td>$I_5$</td>
<td>36</td>
<td>18</td>
<td>$I_4$</td>
<td>47</td>
<td>20</td>
</tr>
</tbody>
</table>
Figure F.2: Measurement plan. Blue lines indicating steel beams. Red rectangles indicating concrete plates building up the walls.
Figure F.3: Wind and other weather conditions at the Kallax weather station during the measurements.
FDD searching in the frequency band 0-50 Hz gave some more modes:

- Mode 1: 1.367 Hz
- Mode 2: 1.904 Hz
- Mode 3: 4.492 Hz
- Mode 4: 5.225 Hz
- Mode 5: 7.983 Hz
- Mode 6: 9.619 Hz
- Mode 7: 10.6 Hz
- Mode 8: 12.25 Hz
- Mode 9: 13.67 Hz
- Mode 10: 14.55 Hz
- Mode 11: 19.07 Hz
- Mode 12: 33.48 Hz
- Mode 13: 40.43 Hz

Mode 10-11 are small variations of the 13.67 Hz mode above.

FDD searching in the frequency band 0-25 Hz gave some more modes:

- Mode 1: 1.335 Hz
- Mode 2: 1.904 Hz
- Mode 3: 6.152 Hz
- Mode 4: 6.234 Hz
- Mode 5: 8.887 Hz
- Mode 6: 9.510 Hz
- Mode 7: 10.6 Hz
- Mode 8: 11.67 Hz
- Mode 9: 12.24 Hz
- Mode 10: 13.42 Hz
- Mode 11: 13.66 Hz
- Mode 12: 14.22 Hz

Figure F.4: FDD modal analysis results.
EFDD Searcing in the frequency band 0-50 Hz (64 bits, 17 h for measurement setup I)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFDD Mode 1</td>
<td>1.365</td>
<td>0.03275</td>
<td>3.403</td>
<td>0.7842</td>
</tr>
<tr>
<td>EFDD Mode 2</td>
<td>1.9</td>
<td>0.009257</td>
<td>1.441</td>
<td>0.3517</td>
</tr>
<tr>
<td>EFDD Mode 3</td>
<td>4.475</td>
<td>0.06582</td>
<td>1.758</td>
<td>0.736</td>
</tr>
<tr>
<td>EFDD Mode 4</td>
<td>5.451</td>
<td>0.4102</td>
<td>2.018</td>
<td>1.682</td>
</tr>
<tr>
<td>EFDD Mode 5</td>
<td>7.991</td>
<td>0.02558</td>
<td>0.217</td>
<td>0.1277</td>
</tr>
<tr>
<td>EFDD Mode 6</td>
<td>9.595</td>
<td>0.03515</td>
<td>0.81</td>
<td>0.2064</td>
</tr>
<tr>
<td>EFDD Mode 7</td>
<td>10.61</td>
<td>0.05682</td>
<td>1.594</td>
<td>0.5216</td>
</tr>
<tr>
<td>EFDD Mode 8</td>
<td>12.28</td>
<td>0.04222</td>
<td>0.7443</td>
<td>0.4059</td>
</tr>
<tr>
<td>EFDD Mode 9</td>
<td>13.67</td>
<td>0.03161</td>
<td>0.248</td>
<td>0.1281</td>
</tr>
<tr>
<td>EFDD Mode 10</td>
<td>14.53</td>
<td>0.05446</td>
<td>0.4520</td>
<td>0.1944</td>
</tr>
<tr>
<td>EFDD Mode 11</td>
<td>19.09</td>
<td>0.1395</td>
<td>0.4742</td>
<td>0.369</td>
</tr>
<tr>
<td>EFDD Mode 12</td>
<td>39.19</td>
<td>0.8541</td>
<td>0.3074</td>
<td>0.497</td>
</tr>
<tr>
<td>EFDD Mode 13</td>
<td>40.52</td>
<td>0.09119</td>
<td>0.3054</td>
<td>0.2395</td>
</tr>
</tbody>
</table>

Measurement errors in measurement point C2 (151).

Figure F.5: EFDD modal analysis results 1.
EFDD Searcing in the frequency band 0-16.67 Hz gave mainly the same modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFDD Mode 1</td>
<td>6.075</td>
<td>0.034234</td>
<td>3.259</td>
<td>0.8051</td>
</tr>
<tr>
<td>EFDD Mode 2</td>
<td>6.234</td>
<td>0.01018</td>
<td>1.138</td>
<td>0.4267</td>
</tr>
<tr>
<td>EFDD Mode 3</td>
<td>6.075</td>
<td>0.2169</td>
<td>0.5633</td>
<td>0.6585</td>
</tr>
<tr>
<td>EFDD Mode 4</td>
<td>6.234</td>
<td>0.004421</td>
<td>0.1981</td>
<td>0.03195</td>
</tr>
<tr>
<td>EFDD Mode 5</td>
<td>8.94</td>
<td>0.1055</td>
<td>1.058</td>
<td>0.8898</td>
</tr>
<tr>
<td>EFDD Mode 6</td>
<td>13.63</td>
<td>0.04294</td>
<td>0.6384</td>
<td>0.2309</td>
</tr>
<tr>
<td>EFDD Mode 7</td>
<td>13.63</td>
<td>0.09566</td>
<td>1.101</td>
<td>0.593</td>
</tr>
<tr>
<td>EFDD Mode 8</td>
<td>11.63</td>
<td>0.08855</td>
<td>0.2576</td>
<td>0.2771</td>
</tr>
<tr>
<td>EFDD Mode 9</td>
<td>12.29</td>
<td>0.07839</td>
<td>0.5094</td>
<td>0.3575</td>
</tr>
<tr>
<td>EFDD Mode 10</td>
<td>13.38</td>
<td>0.1199</td>
<td>0.2339</td>
<td>0.2085</td>
</tr>
<tr>
<td>EFDD Mode 11</td>
<td>13.63</td>
<td>0.08439</td>
<td>0.1472</td>
<td>0.08698</td>
</tr>
<tr>
<td>EFDD Mode 12</td>
<td>14.23</td>
<td>0.05483</td>
<td>0.0849</td>
<td>0.01613</td>
</tr>
</tbody>
</table>

Like the 5.451 Hz mode in the 0-50 Hz search but without the error in point 151.

EFDD Searcing in the frequency band 0-25 Hz gave mainly the same modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFDD Mode 1</td>
<td>9.931</td>
<td>0.055216</td>
<td>3.608</td>
<td>0.8171</td>
</tr>
<tr>
<td>EFDD Mode 2</td>
<td>10.6</td>
<td>0.05634</td>
<td>1.295</td>
<td>0.4923</td>
</tr>
<tr>
<td>EFDD Mode 3</td>
<td>10.6</td>
<td>0.05634</td>
<td>0.6684</td>
<td>1.056</td>
</tr>
<tr>
<td>EFDD Mode 4</td>
<td>11.2</td>
<td>0.1593</td>
<td>0.4533</td>
<td>0.2769</td>
</tr>
<tr>
<td>EFDD Mode 5</td>
<td>12.27</td>
<td>0.06743</td>
<td>0.5905</td>
<td>0.354</td>
</tr>
<tr>
<td>EFDD Mode 6</td>
<td>13.4</td>
<td>0.1426</td>
<td>0.2797</td>
<td>0.2048</td>
</tr>
<tr>
<td>EFDD Mode 7</td>
<td>13.65</td>
<td>0.1039</td>
<td>0.1955</td>
<td>0.09938</td>
</tr>
<tr>
<td>EFDD Mode 8</td>
<td>14.23</td>
<td>0.0658</td>
<td>0.9709</td>
<td>0.01582</td>
</tr>
<tr>
<td>EFDD Mode 9</td>
<td>14.62</td>
<td>0.1368</td>
<td>0.6305</td>
<td>0.0317</td>
</tr>
</tbody>
</table>

EFDD Searcing in the frequency band 0-12.5 Hz gave one new mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFDD Mode 1</td>
<td>1.077</td>
<td>0.0332</td>
<td>2.916</td>
<td>1.184</td>
</tr>
<tr>
<td>EFDD Mode 2</td>
<td>1.9</td>
<td>0.009863</td>
<td>1.108</td>
<td>0.427</td>
</tr>
<tr>
<td>EFDD Mode 3</td>
<td>9.072</td>
<td>0.2266</td>
<td>0.5268</td>
<td>0.6222</td>
</tr>
<tr>
<td>EFDD Mode 4</td>
<td>8.192</td>
<td>0.1265</td>
<td>0.395</td>
<td>1.384</td>
</tr>
<tr>
<td>EFDD Mode 5</td>
<td>8.254</td>
<td>0.1255</td>
<td>0.9853</td>
<td>0.4283</td>
</tr>
<tr>
<td>EFDD Mode 6</td>
<td>8.832</td>
<td>0.07128</td>
<td>0.5059</td>
<td>0.5425</td>
</tr>
</tbody>
</table>

Figure F.6: EFDD modal analysis results 2.
### CFDD Searching in the frequency band 0-50 Hz (32 bits, 17 h for measurement setup I)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFDD Mode 1</td>
<td>1.366</td>
<td>0.03148</td>
<td>2.623</td>
<td>0.558</td>
</tr>
<tr>
<td>CFDD Mode 2</td>
<td>1.902</td>
<td>0.03327</td>
<td>1.435</td>
<td>0.2708</td>
</tr>
<tr>
<td>CFDD Mode 3</td>
<td>4.469</td>
<td>0.07128</td>
<td>1.038</td>
<td>0.3145</td>
</tr>
<tr>
<td>CFDD Mode 4</td>
<td>5.4</td>
<td>0.3154</td>
<td>1.435</td>
<td>0.7192</td>
</tr>
<tr>
<td>CFDD Mode 5</td>
<td>7.983</td>
<td>0.03894</td>
<td>0.1455</td>
<td>0.03333</td>
</tr>
<tr>
<td>CFDD Mode 6</td>
<td>9.592</td>
<td>0.03849</td>
<td>0.392</td>
<td>0.1419</td>
</tr>
<tr>
<td>CFDD Mode 7</td>
<td>10.6</td>
<td>0.05032</td>
<td>0.8076</td>
<td>0.2834</td>
</tr>
<tr>
<td>CFDD Mode 8</td>
<td>12.25</td>
<td>0.05113</td>
<td>0.4411</td>
<td>0.1967</td>
</tr>
<tr>
<td>CFDD Mode 9</td>
<td>13.39</td>
<td>0.1395</td>
<td>0.2001</td>
<td>0.2447</td>
</tr>
<tr>
<td>CFDD Mode 10</td>
<td>13.65</td>
<td>0.07795</td>
<td>0.1398</td>
<td>0.0579</td>
</tr>
<tr>
<td>CFDD Mode 11</td>
<td>14.53</td>
<td>0.05588</td>
<td>0.2175</td>
<td>0.1074</td>
</tr>
<tr>
<td>CFDD Mode 12</td>
<td>19.09</td>
<td>0.1389</td>
<td>0.2311</td>
<td>0.2057</td>
</tr>
<tr>
<td>CFDD Mode 13</td>
<td>39.48</td>
<td>0.1045</td>
<td>0.1251</td>
<td>0.1041</td>
</tr>
<tr>
<td>CFDD Mode 14</td>
<td>40.49</td>
<td>0.1067</td>
<td>0.1625</td>
<td>0.1288</td>
</tr>
</tbody>
</table>

Measurement errors in measurement points C2 (151) here and for other analysis methods. The corresponding errors in measurement points C3,C4,C5 (152,192,193) are not present in the EFDD results, so they are mainly dependent on the method.

Figure F.7: CFDD modal analysis results 1.
CFDD Searcing in the frequency band 0-25 Hz gave some extra modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFDD Mode 1</td>
<td>1.365</td>
<td>0.03228</td>
<td>2.34</td>
<td>0.5076</td>
</tr>
<tr>
<td>CFDD Mode 2</td>
<td>1.901</td>
<td>0.00509</td>
<td>1.013</td>
<td>0.3002</td>
</tr>
<tr>
<td>CFDD Mode 3</td>
<td>5.972</td>
<td>0.3475</td>
<td>0.4778</td>
<td>0.6104</td>
</tr>
<tr>
<td>CFDD Mode 4</td>
<td>6.232</td>
<td>0.05303</td>
<td>0.1055</td>
<td>0.0419</td>
</tr>
<tr>
<td>CFDD Mode 5</td>
<td>9.584</td>
<td>0.06393</td>
<td>0.3273</td>
<td>0.1748</td>
</tr>
<tr>
<td>CFDD Mode 6</td>
<td>9.933</td>
<td>0.01004</td>
<td>0.38147</td>
<td>0.0514</td>
</tr>
<tr>
<td>CFDD Mode 7</td>
<td>10.81</td>
<td>0.05452</td>
<td>0.7951</td>
<td>0.3401</td>
</tr>
<tr>
<td>CFDD Mode 8</td>
<td>11.195</td>
<td>0.1601</td>
<td>0.2423</td>
<td>0.1632</td>
</tr>
<tr>
<td>CFDD Mode 9</td>
<td>12.245</td>
<td>0.05177</td>
<td>0.411</td>
<td>0.1945</td>
</tr>
<tr>
<td>CFDD Mode 10</td>
<td>13.44</td>
<td>0.1445</td>
<td>0.1349</td>
<td>0.1209</td>
</tr>
<tr>
<td>CFDD Mode 11</td>
<td>13.645</td>
<td>0.1035</td>
<td>0.1125</td>
<td>0.06256</td>
</tr>
<tr>
<td>CFDD Mode 12</td>
<td>14.235</td>
<td>0.005655</td>
<td>0.05139</td>
<td>0.01384</td>
</tr>
<tr>
<td>CFDD Mode 13</td>
<td>14.555</td>
<td>0.0784</td>
<td>0.1771</td>
<td>0.1423</td>
</tr>
</tbody>
</table>

Figure F.8: CFDD modal analysis results 2.

CFDD Searcing in the frequency band 0-12.5 Hz gave some extra modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFDD Mode 1</td>
<td>1.363</td>
<td>0.02933</td>
<td>2.055</td>
<td>0.6158</td>
</tr>
<tr>
<td>CFDD Mode 2</td>
<td>1.901</td>
<td>0.01055</td>
<td>0.8871</td>
<td>0.34</td>
</tr>
<tr>
<td>CFDD Mode 3</td>
<td>6.669</td>
<td>0.2367</td>
<td>0.321</td>
<td>0.4211</td>
</tr>
<tr>
<td>CFDD Mode 4</td>
<td>6.235</td>
<td>0.004134</td>
<td>0.1053</td>
<td>0.02405</td>
</tr>
<tr>
<td>CFDD Mode 5</td>
<td>7.978</td>
<td>0.05166</td>
<td>0.3337</td>
<td>0.4022</td>
</tr>
<tr>
<td>CFDD Mode 6</td>
<td>8.252</td>
<td>0.1285</td>
<td>0.5707</td>
<td>0.2909</td>
</tr>
<tr>
<td>CFDD Mode 7</td>
<td>8.812</td>
<td>0.06872</td>
<td>0.3889</td>
<td>0.3726</td>
</tr>
</tbody>
</table>
F.1 Modal analysis

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSI-UPC Mode 1</td>
<td>40</td>
<td>0.02376</td>
<td>0.2923</td>
<td>0.1684</td>
</tr>
</tbody>
</table>

SSI-UPC, searching in the frequency band 0-10 Hz

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSI-UPC Mode 1</td>
<td>1.902</td>
<td>0.009823</td>
<td>1.486</td>
<td>0.4506</td>
</tr>
<tr>
<td>SSI-UPC Mode 2</td>
<td>5.373</td>
<td>0.1211</td>
<td>3.531</td>
<td>0.5482</td>
</tr>
<tr>
<td>SSI-UPC Mode 3</td>
<td>9.581</td>
<td>0.04799</td>
<td>1.577</td>
<td>0.267</td>
</tr>
</tbody>
</table>

SSI-PC, searching in the frequency band 0-50 Hz

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSI-PC Mode 1</td>
<td>39.97</td>
<td>0.04495</td>
<td>0.3372</td>
<td>0.2848</td>
</tr>
</tbody>
</table>

SSI-PC, searching in the frequency band 0-10 Hz

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSI-PC Mode 1</td>
<td>1.9</td>
<td>0.009444</td>
<td>1.502</td>
<td>0.5149</td>
</tr>
<tr>
<td>SSI-PC Mode 2</td>
<td>9.605</td>
<td>0.06571</td>
<td>1.787</td>
<td>0.3304</td>
</tr>
</tbody>
</table>

Figure F.9: SSI-UPC and SSI-PC modal analysis results.
SSI-CVA, searching in the frequency band 0-50 Hz

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSI-CVA Mode 1</td>
<td>40</td>
<td>0.02821</td>
<td>0.1824</td>
<td>0.2022</td>
</tr>
</tbody>
</table>

SSI-CVA, searching in the frequency band 0-25 Hz

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSI-CVA Mode 1</td>
<td>1.366</td>
<td>0.02914</td>
<td>2.866</td>
<td>0.7058</td>
</tr>
<tr>
<td>SSI-CVA Mode 2</td>
<td>1.9</td>
<td>0.01052</td>
<td>1.746</td>
<td>0.6220</td>
</tr>
<tr>
<td>SSI-CVA Mode 3</td>
<td>4.452</td>
<td>0.05438</td>
<td>1.563</td>
<td>0.3455</td>
</tr>
<tr>
<td>SSI-CVA Mode 4</td>
<td>5.369</td>
<td>0.1279</td>
<td>2.348</td>
<td>0.6791</td>
</tr>
<tr>
<td>SSI-CVA Mode 5</td>
<td>8.186</td>
<td>0.138</td>
<td>2.73</td>
<td>0.4013</td>
</tr>
</tbody>
</table>

SSI-CVA, searching in the frequency band 0-12.5 Hz also gave a mode at 9.59 Hz

<table>
<thead>
<tr>
<th>Mode</th>
<th>Frequency [Hz]</th>
<th>Std. Frequency [Hz]</th>
<th>Damping Ratio [%]</th>
<th>Std. Damping Ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSI-CVA Mode 1</td>
<td>1.364</td>
<td>0.02729</td>
<td>2.457</td>
<td>0.6504</td>
</tr>
<tr>
<td>SSI-CVA Mode 2</td>
<td>1.901</td>
<td>0.009401</td>
<td>1.253</td>
<td>0.5423</td>
</tr>
<tr>
<td>SSI-CVA Mode 3</td>
<td>4.456</td>
<td>0.05355</td>
<td>1.568</td>
<td>0.2223</td>
</tr>
<tr>
<td>SSI-CVA Mode 4</td>
<td>5.367</td>
<td>0.1496</td>
<td>2.764</td>
<td>0.9362</td>
</tr>
<tr>
<td>SSI-CVA Mode 5</td>
<td>9.59</td>
<td>0.05568</td>
<td>1.524</td>
<td>0.2451</td>
</tr>
</tbody>
</table>

Low signal-to-noise-ratio (SNR) at some low frequencies at measurement point C2 (151). Probably no easy fix, since time-frequency analysis indicates that the SNR at those frequencies is roughly the same during the whole measurement time.

Figure F.10: SSI-CVA modal analysis results.
• Mode 8 only found by EFDD, CFDD and SSI-CVA?
• Mode 19 only found by SSI-UPC and SSI-PC?
• FDD finds all the others, but for some modes, a smoother and seemingly more realistic modeshapes are found with other methods.
• In some cases, we have interpreted similar mode shapes at very close frequencies as the same. For example, in Figure F.8 CFDD seems to do “double detection” of modes 10 and 14 on nearby frequencies. Comparison with modes predicted by our FE model will show if any such closely located modes are different, since the FE model only predicts modes whose mode shapes are orthogonal with the inner product given by the mass matrix in (I.18) (page 102).

F.2 FE models

For the tower, we also made finite element models in the software Comsol, which is integrated with MATLAB, contrary to the Abaqus FE software. Therefore we suspected that it might be easier for the FEM updating (FEMU) software to interact with Comsol than with Abaqus. Due to no previous experience with Comsol, we only tried making models in Comsol in this project, and then made the same models in Abaqus, as a validation of the Comsol models.

We built two version of both models and one third model in Abaqus.

Version I A pure concrete shear-wall Comsol and Abaqus model containing the walls but not the steel beams.

Version II An improvement of the version I models with steel beams added.

Version III A third model (Abaqus only), with new boundary conditions, with Floor 2 fixed to the ground as well as floor four and below fixed to the neighbouring building.

Figure F.11 shows a comparison of predicted mode shapes and frequencies with the corresponding measured modal data. Eight mode shapes could be matched to corresponding measured mode shapes with similar frequencies. For closer matching frequencies, especially for the second mode, we think that further improvement of the models can be needed, for example by adding the brick walls that are covering the outside of the concrete walls. We also think that a larger number of predicted modes would have been found in measurements if there had been stronger wind during the measurements.
Figure F.11: Comparison of predicted and measured modal data.
G Kirchhoff plate — convex optimization without FE software interaction

For demonstrating how to run FEM-updating for another software than Abaqus and for demonstrating how to use convex optimization instead of the Newton trust-region approximation used in previous versions of the software, we implemented a simulated Kirchhoff plate similar to the damage detection demonstrated in [Her14], where point damages were detected based only on frequency residuals. The main generalizations that we have done so far are the following:

- Damage detection based on both frequency and mode shape residuals. (For older results based only on frequency residuals, see the paper [HGS+16], which is included as Appendix L, starting on page 268)

- Generalizations for detection also of larger damages in a scenario where forces and moments on the plate are such that damages caused by the load primarily are cracks that are propagating in a direction close to some known angle.

The simulations of this section are optimized for cracks propagating mainly in a direction parallel to the diagonal from the lower left to the upper right corner of the plate, as for the big damage in Figure G.3 below, but the angle is an input parameter in the MATLAB code, so there is no restriction to angle 45 degrees there.

Such larger damages are more realistic, for instance, for concrete plates that are part of a bigger structure. See Section J.4.2 for the Kirchhoff plate source code.

A convex formulation of the optimization problem can also be useful for large structures such as the bridges in this paper, where it might be of importance that convex optimization methods can use the convex formulation of the problem to speed up computations.

G.1 How to run the software

The code specific for the kirchhoff plate is in the directory SHM/FEMU/Impl/SimulatedPlate. Before running the Main-file, you need to add the path SHM/FEMU/Library to the matlab path. No Abaqus or other FEM software is needed. You also need to download and install the CVX package for convex optimization, which is available for free from http://cvxr.com/cvx/. The following are some of the parameters that can be changed.

Running in parallel on several CPU cores Previous versions could in theory run from separate main-files in different MATLAB windows with some duplication of output files, but it did not work so well in practice, because when calling Abaqus from different MATLAB windows, computations were often interrupted by an Abaqus crash before converging to a solution.

This problem disappeared when we replaced the AbaqusPkg package with the smaller MatlabSimPkg package, which also both removes the need for writing, reading and parsing text files, as well as resulting roundoff errors that could increase the risk for numerical problems.

As a preparation of running in parallel in eight different MATLAB windows, the directory SHM/FEMU/Impl/SimulatedPlate contains the directories KirchhA, KirchhB, KirchhC, ..., KirchhH, which contain files with the input parameters and output files that are local for the main file in the same directory. In the following, unless otherwise stated, we refer to files in this directory.
Run the Main-file of the current directory to run damage detection on the Kirchoff plate with default settings.

**Changing damages, number of modes and/or adding noise** The file README.txt explains how to use the file simulateDamage.m to set the element numbers and sizes of the desired damages. The elements are numbered as in Figure G.1.

The resulting three lines for the point damage and the big damage in Figure G.3 below are already in the file InputParameters.m.

Comments in simulateDamage.m also tells how noise can be added, and you can change other parameters, such as the dimensions of the plate.

**Regularization method** In section 27 of the main file, you can choose regularization technique. For the ones described in this appendix, set dictStr to either 'KronDelta', 'haar' or 'AllCracksTheta'.

**Measurement points are chosen automatically** as indicated in Figure G.2 by multiplying the chosen measured mode shapes and do measurements in the points where this product has its largest local maxima.

**Results of the damage detection** are saved in the subdirectory Results.

See the comments in the Main-file, InputParameters.m and simulateDamage.m for more examples of parameters that can be changed.

### G.2 Preliminary results

For the optimization problem (I.47) described in Section I.4.2 we have run simulations for a point damage and a big damage, for which the damage index is plotted in the leftmost column of Figure (G.3). The undamaged elements have elastic modulus $E_0 = 200$ GPa. For the point damage, the damaged element has elastic modulus 180 GPa. For the big damage, elastic modulus is 180 GPa along the main diagonal and 190 GPa for the other damaged elements. (Corresponding to a long deep crack, some less deep shorter cracks in different directions and a point damage.) The plotted damage index for an element with elastic modulus $E_j$ is $DI = \frac{E_0 - E_j}{E_0}$.

For the three different dictionaries and damage detection with shape residual weight $W_s$ and regularization term weight $\lambda$ given by equation (I.47), we have run simulations for a larger number of different combinations of the parameters $W_s$ and $\lambda$, as described in more detail in separate subsection below. Some of the best damage detections obtained for the different dictionaries are summarized in column 2–4 of Figure (G.3). The MATLAB code for these methods is still in an experimental stage, so these are still preliminary results that might improve after more fine tuning of the implemented optimization methods.

The Kronecker Delta dictionary gives the damage detection results in column 2 of Figure (G.3). Perfect identification of the point damage but only partial detection of the big damage. Partial indication of the long deep crack and of the smaller vertical and horizontal cracks, and no indication of the other damages. (For $W_s = 0$, this method corresponds to the one in HGS+16, for which we describe some "ghost image" problems and presented corresponding results in HGS+16, see subsection 3.4.2 of Appendix L.)

The third column in Figure (G.3) shows that the Haar dictionary still gives a rather precise detection of the size and localization of the point damage, but a more precise indication of
the big damage. Better indication of the long deep crack and still partial indication of smaller vertical and horizontal cracks, but still no indication of the other damages.

Figure G.1: Element numbers (orange) and node numbers (yellow) for the Kirchhoff plate (361 internal nodes).

Figure G.2: Product of the 10 used mode shapes plotted against the node numbers with numbering as in Figure G.1 and with a point damage in element 211, where the Young modulus was reduced 10% from 200 to 180 GPa. The eight largest peaks, marked with red stars, are at node number 92, 98, 152, 278, 290, 344 and 350. Element and node numbering. These points are chosen as measurement points.
For the 45° Cracks dictionary, the rightmost column in Figure (G.3) indicates the right total size for the point damage, but with localization spread out to two elements. For the big damage, however, we now get relatively precise detection of the size and localization of both the long deep crack and of the shorter less deep cracks parallel to it. Also similar partial indication of the vertical and horizontal cracks and still no detection of the small point damage.

The following are some potential topics for further development and research:

- For these results we had to make visual comparisons to decide which combinations of the parameters $W_s$ and $\lambda$ to use. For more practical use it would be quite desirable to have some kind of automatic choice of these parameters. One possible approach could be to try using an L-curve approach as in Figure 15 of Appendix K.

- The MATLAB code is still in an experimental stage. For example, some further fine tuning of the implementation of Nelson’s method might give improved damage detection.

Another example can be seen especially in Figure (G.7) below: Some combinations of damages and parameter values can give non-numerical values in the stiffness matrix, which indicates that there is some special case not yet covered in the implementation. (Indicated with white text in the corresponding plots.) This shall of course be further investigated.

- We have used one dictionary (Haar) of nearly minimal size and one quite overcomplete dictionary (45° Cracks). One idea for a way to construct another smaller dictionary could be to run a dictionary learning algorithm with the 45° Cracks dictionary as input, to get another suggestion for a smaller dictionary to try using.

- Many of the damage detection results in the following subsections could be computed in less than an hour, and the plots in Figure (G.3) are among the fast one, with each one requiring only 8-21 iterative calls of the CVX convex optimization software. For some other combinations of damage, $W_s$ and $\lambda$, however, the algorithm did not converge in

![Figure G.3: Damage index for a point damage, a big damage as well as detected damages for the convex optimization problem (I.47) on page 114 with three different dictionaries. Here $\lambda$ is the regularization parameter in (I.47), the frequency residual weights in (I.9) on page 99 equals 1 and all mode shape residuals equals $W_s$.](image-url)
reasonable time. For two of the plots indicated in Figure G.6 and a few more not included below, the computations were interrupted after over 6000 CVX calls, which in these cases meant three weeks of CPU-time for each plot. This is far beyond reasonable, so after that we added an automatic interruption after 1000 CVX calls, which often corresponded to about 48 h of CPU time. It is indicated with yellow text in the figures below, which plots that needed that was interrupted after 1000 CVX calls. If some improvements in, for example, the Nelson implementation can reduce the time of convergence in these cases, that would of course be great.

The following subsections describe in more detail the work that was needed to obtain the results in Figure G.3.

**Damage detection of the point damage in Figure G.3**

For parameters $\lambda$ and $W_s$ as described in Figure G.3, Figure G.4 shows nearly perfect detection of the point damage for $1/8 \leq W_s \leq 2$. Each row shows the computed damage index for regularization weight parameter values

$$\lambda = 7.5 \cdot 10^{-8}, 1 \cdot 10^{-7}, 2.5 \cdot 10^{-7}, 5 \cdot 10^{-7}, 7.5 \cdot 10^{-7}, 1 \cdot 10^{-6}, 2.5 \cdot 10^{-6}, 5 \cdot 10^{-6}.$$  

Figure G.5 show damage detection results for the point damage when using the Haar dictionary. For each $W_s$, the subplots are for the following values of the regularization term weight

$$W_r = 0.125, 0.25, 0.5, 1, 2$$

Figure G.4: Damage detection of the point damage using the Kronecker Delta dictionary.
\[ \lambda: \quad 1 \cdot 10^{-9} \quad 1 \cdot 10^{-8} \quad 2.5 \cdot 10^{-8} \quad 5 \cdot 10^{-8} \quad 7.5 \cdot 10^{-8} \quad 1 \cdot 10^{-7} \]
\[ 2.5 \cdot 10^{-7} \quad 5 \cdot 10^{-7} \quad 7.5 \cdot 10^{-7} \quad 1 \cdot 10^{-6} \quad 2.5 \cdot 10^{-6} \quad 5 \cdot 10^{-6} \]

When \( 1 \leq W_s \leq 2 \) we can see a rather precise localization of the damage for some \( \lambda \). Larger \( W_s \) would make less use of information in the frequency residual, which was expected to give less precise damage detection.

Figure [G.6] shows damage detection results for the point damage when using 45° Cracks dictionary. For each \( W_s \), the subplots are for the following values of the regularization term weight \( \lambda \):

\[ 1 \cdot 10^{-9} \quad 2.5 \cdot 10^{-9} \quad 5 \cdot 10^{-8} \quad 1 \cdot 10^{-7} \quad 2.5 \cdot 10^{-7} \quad 5 \cdot 10^{-7} \quad 7.5 \cdot 10^{-7} \quad 1 \cdot 10^{-6} \quad 2.5 \cdot 10^{-6} \quad 5 \cdot 10^{-6} \]

\( W_s = 0.5 \) (Similar but slightly more spread out for \( W_s = 1/8 \) and \( W_s = 1/4 \).)

![Figure G.5: Damage detection of the point damage using the Haar dictionary.](image-url)
The most precise damage detection was obtained when $1/512 \leq W_s \leq 1/16$.

**Damage detection of the big damage in Figure G.3**

For parameters $\lambda$ and $W_s$ as described in Figure G.3, Figure G.7 shows damage detection results for the big damage when using the Kronecker Delta dictionary. For each $W_s$, the subplots are for the following values of the regularization term weight $\lambda$:

$$
\begin{align*}
1 \cdot 10^{-9} & \quad 2.5 \cdot 10^{-9} & \quad 5 \cdot 10^{-9} & \quad 7.5 \cdot 10^{-9} & \quad 1 \cdot 10^{-8} & \quad 2.5 \cdot 10^{-8} & \quad 5 \cdot 10^{-8} \\
7.5 \cdot 10^{-8} & \quad 1 \cdot 10^{-7} & \quad 2.5 \cdot 10^{-7} & \quad 5 \cdot 10^{-7} & \quad 7.5 \cdot 10^{-7} & \quad 1 \cdot 10^{-6} & \quad 2.5 \cdot 10^{-6}
\end{align*}
$$

Additional plots for the same $W_s$ and $1 \cdot 10^{-12} \leq \lambda \leq 7.5 \cdot 10^{-10}$ are all almost identical to the results for $W_s = 4$ and $\lambda = 1 \cdot 10^{-9}$ and therefore excluded. A few additional plots for $W_s = 1$ and $5 \cdot 10^{-6} \leq \lambda \leq 2.5 \cdot 10^{-5}$ looks similar to the case $W_s = 1$ and $\lambda = 2.5 \cdot 10^{-6}$. Thus we saw no reason to expect more precise damage detection for larger or smaller $\lambda$. As expected, these results indicate that the Kronecker Delta dictionary is really good for point damages, but the sparsity that is forced onto the solution by the regularization term makes this dictionary less suitable for larger damages.

Since point damages were indicated with the Haar dictionary for $W_s = 1$ and $W_s = 2$, but not for $W_s = 0.5$, we started from the assumption that $W_s = 1$ and $W_s = 2$ should be a good choice also for big damages, whereas larger $W_s$ makes less use of the frequency residual, which can be expected to give less sensitive damage detection. Figure G.8 shows the damage detection results for the big damage with the Haar dictionary. For each $W_s$, the subplots are for the following values of the regularization term weight $\lambda$:

$$
\begin{align*}
1 \cdot 10^{-9} & \quad 2.5 \cdot 10^{-9} & \quad 5 \cdot 10^{-9} & \quad 7.5 \cdot 10^{-9} & \quad 1 \cdot 10^{-8} & \quad 2.5 \cdot 10^{-8} & \quad 5 \cdot 10^{-8} \\
7.5 \cdot 10^{-8} & \quad 1 \cdot 10^{-7} & \quad 2.5 \cdot 10^{-7} & \quad 5 \cdot 10^{-7} & \quad 7.5 \cdot 10^{-7} & \quad 1 \cdot 10^{-6} & \quad 2.5 \cdot 10^{-6} \\
5 \cdot 10^{-6} & \quad 7.5 \cdot 10^{-6} & \quad 1 \cdot 10^{-5} & \quad 2.5 \cdot 10^{-5} & \quad 5 \cdot 10^{-5} & \quad 7.5 \cdot 10^{-5} & \quad 1 \cdot 10^{-4}
\end{align*}
$$

(The first two rows are identical to (G.1).)

For damage detection of a point damage with the 45° Cracks dictionary in last section, we got the best localization of the damage for $1/512 \leq W_s \leq 1/16$. Therefore, we first restricted to $1/512 \leq W_s \leq 1/8$ for the big damage, with results overview in Figure G.9. For each $W_s$ the subplots are again for the $\lambda$ given in (G.1). However, much to our surprise, we got more precise localization of the damages when trying larger $W_s$, as shown for the same $\lambda$ in Figure G.9.

**Point damage, minimize coefficients of $\Delta a^i$ only**

For completeness, we also evaluated the regularization problem (I.46) with a regularization term that strives for sparse coefficients $c$ in the composition $\Delta a^i = \Psi c$ rather than sparse coefficients $b$ in the decomposition $a^i = \Psi b$. However, as suspected, this only gave much slower convergence to much less precise damage detection, as shown for the point damage and the 45° Cracks dictionary in Figure G.11.
$W_1 = 1/512$ (Nearly identical for $W_1 = 1/256$, $W_1 = 1/128$, $W_1 = 1/64$, and $W_1 = 1/32$.)

$W_1 = 1/16 = 0.0625$

$W_1 = 1/8 = 0.125$

$W_1 = 1/4 = 0.25$

$W_1 = 1$(Nearly identical and only slightly more spread out for $W_1 = 2$.)

Figure G.6: Damage detection of the point damage using the 45° Cracks dictionary.
Figure G.7: Damage detection of the big damage using the Kronecker Delta dictionary.
Figure G.8: Damage detection of the big damage using the Haar dictionary. We also did the same plots for $W_s = 4$ and $W_s = 8$, but they contained no improvements and are therefore excluded here.
$W_i = 1/64 = 0.015625$ (similar but more spread out for $W_i = 1/128$ and $W_i = 1/512$.)

$W_i = 1/32 = 0.03125$

$W_i = 1/16 = 0.0625$

$W_i = 1/8 = 0.125$

Figure G.9: Damage detection of the big damage using the 45° Cracks dictionary with $1/512 \leq W_s \leq 1/8$. 
Figure G.10: Damage detection of the big damage using the 45° Cracks dictionary with $1/2 \leq W \leq 8$. 
Figure G.11: Damage detection of the point damage using the 45° Cracks dictionary for the regularization problem [1.46] on page 114.
Table H.1: Calibration parameters for accelerometer 1.

<table>
<thead>
<tr>
<th>Table values</th>
<th>$a_1$ (rad)</th>
<th>$a_2$ (rad)</th>
<th>$a_3$ (rad)</th>
<th>$b_1$ (mV)</th>
<th>$b_2$ (mV)</th>
<th>$b_3$ (mV)</th>
<th>$g_1$ (mV/g)</th>
<th>$g_2$ (mV/g)</th>
<th>$g_3$ (mV/g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Åby River '12</td>
<td>-0.0046642</td>
<td>0.0009415</td>
<td>-0.0099966</td>
<td>157.5692</td>
<td>-190.552</td>
<td>-163.9992</td>
<td>1269.3104</td>
<td>1274.282</td>
<td>1274.3582</td>
</tr>
<tr>
<td>Åby River '13</td>
<td>0.000664</td>
<td>-0.003664</td>
<td>0.001008</td>
<td>-106.1567</td>
<td>-163.6431</td>
<td>-165.1062</td>
<td>1266.7315</td>
<td>1274.6083</td>
<td>1274.3582</td>
</tr>
<tr>
<td>kiruna '14a</td>
<td>-0.000664</td>
<td>0.000000</td>
<td>-0.000000</td>
<td>-104.323</td>
<td>-158.9877</td>
<td>-15.144</td>
<td>1262.1243</td>
<td>1227.4845</td>
<td>1227.4845</td>
</tr>
<tr>
<td>kiruna '14b</td>
<td>0.000000</td>
<td>0.003100</td>
<td>-0.000000</td>
<td>-107.5558</td>
<td>-163.1062</td>
<td>-206.0674</td>
<td>1264.3801</td>
<td>1274.2841</td>
<td>1274.2841</td>
</tr>
<tr>
<td>kiruna '14c</td>
<td>-0.000100</td>
<td>0.000000</td>
<td>-0.000000</td>
<td>-105.1567</td>
<td>-158.9877</td>
<td>-15.144</td>
<td>1262.1243</td>
<td>1227.4845</td>
<td>1227.4845</td>
</tr>
</tbody>
</table>

The tables also show that the values differ slightly from table values shipped with the accelerometers. Thus we expect to have performed a better elimination of accelerometer imperfections in our results than if we had used the table values in our analysis.

For the Långforsen and Åby river measurements, exactly the same twisted pair wires were connected to each accelerometer. For the Kiruna measurements there were some random changes in the cables used. Since the calibrated parameters describe properties of the whole calibrated system, these changes of cables can, together with temperature changes, be partly responsible for the small visible parameter changes between different measurement occasions.

Due to these differences in parameters, we expect to have performed a better elimination of accelerometer imperfections in our results than if we had used the table values only in our analysis.

The accelerometers 2 and 5 were dismantled and damaged during other measurements 2013, so that no y-direction measurements were possible after that. The calibration methods only work for triaxial accelerometers, so table values were therefore used in the Åby '3 and Kiruna measurements with accelerometers 2 and 5.

Table H.2: Calibration parameters for accelerometer 2.

<table>
<thead>
<tr>
<th>Table values</th>
<th>$a_1$ (rad)</th>
<th>$a_2$ (rad)</th>
<th>$a_3$ (rad)</th>
<th>$b_1$ (mV)</th>
<th>$b_2$ (mV)</th>
<th>$b_3$ (mV)</th>
<th>$g_1$ (mV/g)</th>
<th>$g_2$ (mV/g)</th>
<th>$g_3$ (mV/g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Åby River '12</td>
<td>0.000967</td>
<td>0.0017966</td>
<td>-0.013579</td>
<td>-178.3438</td>
<td>-147.11</td>
<td>-28.1363</td>
<td>1262.8396</td>
<td>1284.7284</td>
<td>1274.7803</td>
</tr>
<tr>
<td>Åby River '13</td>
<td>0.01192</td>
<td>-0.01780</td>
<td>-0.00799</td>
<td>-134.0017</td>
<td>-212.1655</td>
<td>-215.2547</td>
<td>1264.9371</td>
<td>1251.8224</td>
<td>1225.8035</td>
</tr>
<tr>
<td>kiruna '14a</td>
<td>0.00984</td>
<td>-0.01805</td>
<td>-0.00498</td>
<td>-127.7374</td>
<td>-204.91901</td>
<td>-169.8574</td>
<td>1267.6392</td>
<td>1255.0017</td>
<td>1226.2305</td>
</tr>
<tr>
<td>kiruna '14b</td>
<td>0.01111</td>
<td>-0.01964</td>
<td>-0.00341</td>
<td>-129.5786</td>
<td>-207.9565</td>
<td>-167.8321</td>
<td>1265.1294</td>
<td>1252.0187</td>
<td>1225.2635</td>
</tr>
<tr>
<td>kiruna '14c</td>
<td>0.01329</td>
<td>-0.01764</td>
<td>-0.00492</td>
<td>-132.3760</td>
<td>-210.6688</td>
<td>-169.6455</td>
<td>1267.3954</td>
<td>1254.4622</td>
<td>1226.2586</td>
</tr>
</tbody>
</table>

Table H.3: Calibration parameters for accelerometer 3.

<table>
<thead>
<tr>
<th>Table values</th>
<th>$a_1$ (rad)</th>
<th>$a_2$ (rad)</th>
<th>$a_3$ (rad)</th>
<th>$b_1$ (mV)</th>
<th>$b_2$ (mV)</th>
<th>$b_3$ (mV)</th>
<th>$g_1$ (mV/g)</th>
<th>$g_2$ (mV/g)</th>
<th>$g_3$ (mV/g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Åby River '12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-134.495</td>
<td>-213.624</td>
<td>-206.180</td>
<td>1264.4217</td>
<td>1252.0187</td>
<td>1225.2635</td>
</tr>
<tr>
<td>Åby River '13</td>
<td>0.01192</td>
<td>-0.01780</td>
<td>-0.00799</td>
<td>-134.0017</td>
<td>-212.1655</td>
<td>-215.2547</td>
<td>1264.9371</td>
<td>1251.8224</td>
<td>1225.8035</td>
</tr>
<tr>
<td>Kiruna '14a</td>
<td>0.00984</td>
<td>-0.01805</td>
<td>-0.00498</td>
<td>-127.7374</td>
<td>-204.91901</td>
<td>-169.8574</td>
<td>1267.6392</td>
<td>1255.0017</td>
<td>1226.2305</td>
</tr>
<tr>
<td>Kiruna '14b</td>
<td>0.01111</td>
<td>-0.01964</td>
<td>-0.00341</td>
<td>-129.5786</td>
<td>-207.9565</td>
<td>-167.8321</td>
<td>1265.1294</td>
<td>1252.0187</td>
<td>1225.2635</td>
</tr>
<tr>
<td>Kiruna '14c</td>
<td>0.01329</td>
<td>-0.01764</td>
<td>-0.00492</td>
<td>-132.3760</td>
<td>-210.6688</td>
<td>-169.6455</td>
<td>1267.3954</td>
<td>1254.4622</td>
<td>1226.2586</td>
</tr>
</tbody>
</table>

H Accelerometer calibration

All triaxial accelerometers were calibrated for the measurements described in this report. For the Långforsen measurements in Appendix C we used the six-parameter method described in [GS11]. For all other field measurements we used the refined nine-parameter method proposed in [FGS13]. This calibration gives six parameters $b_n, g_n$ that describes the linear dependence of the output voltage on each axis on the measured acceleration, as well as three parameters $a_n$ that describes the amounte of “leakage” between the three measurements caused by electric leakage and by nonorthogonal axes. Table H.1-H.6 show that these parameters no not change much between the five different field measurements done during 2012-2014.

The tables also show that the values differ slightly from table values shipped with the accelerometers. Thus we expect to have performed a better elimination of accelerometer imperfections in our results than if we had used the table values in our analysis.

For the Långforsen and Åby river measurements, exactly the same twisted pair wires were connected to each accelerometer. For the Kiruna measurements there were some random changes in the cables used. Since the calibrated parameters describe properties of the whole calibrated system, these changes of cables can, together with temperature changes, be partly responsible for the small visible parameter changes between different measurement occasions.

Due to these differences in parameters, we expect to have performed a better elimination of accelerometer imperfections in our results than if we had used the table values only in our analysis.
Table H.4: Calibration parameters for accelerometer 4.

<table>
<thead>
<tr>
<th>Table values</th>
<th>α₁ (rad)</th>
<th>α₂ (rad)</th>
<th>α₃ (rad)</th>
<th>b₁ (mV)</th>
<th>b₂ (mV)</th>
<th>b₃ (mV)</th>
<th>g₁ (mV/g)</th>
<th>g₂ (mV/g)</th>
<th>g₃ (mV/g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Åby River 12</td>
<td>0.00046144</td>
<td>-0.019849</td>
<td>-0.0074749</td>
<td>-194.7623</td>
<td>-96.6848</td>
<td>-101.9177</td>
<td>1272.3554</td>
<td>1268.3381</td>
<td>1274.7627</td>
</tr>
<tr>
<td>Åby River 13</td>
<td>0.00272</td>
<td>-0.01872</td>
<td>-0.00779</td>
<td>-202.95612</td>
<td>-100.9129</td>
<td>-110.28763</td>
<td>1277.0271</td>
<td>1273.5411</td>
<td>1260.4204</td>
</tr>
<tr>
<td>Kiruna 14a</td>
<td>-0.00004</td>
<td>-0.01962</td>
<td>-0.00748</td>
<td>-194.75185</td>
<td>-91.11081</td>
<td>-103.05991</td>
<td>1271.6284</td>
<td>1268.5588</td>
<td>1257.2577</td>
</tr>
<tr>
<td>Kiruna 14b</td>
<td>0.00251</td>
<td>-0.01926</td>
<td>-0.008053</td>
<td>-196.18571</td>
<td>-98.39925</td>
<td>-101.03881</td>
<td>1277.30271</td>
<td>1274.59998</td>
<td>1263.1138</td>
</tr>
<tr>
<td>Kiruna 14c</td>
<td>-0.00135</td>
<td>-0.01916</td>
<td>-0.008053</td>
<td>-197.98439</td>
<td>-96.8125</td>
<td>-104.9177</td>
<td>1276.04824</td>
<td>1273.8818</td>
<td>1260.0763</td>
</tr>
</tbody>
</table>

Table H.5: Calibration parameters for accelerometer 5.

<table>
<thead>
<tr>
<th>Table values</th>
<th>α₁ (rad)</th>
<th>α₂ (rad)</th>
<th>α₃ (rad)</th>
<th>b₁ (mV)</th>
<th>b₂ (mV)</th>
<th>b₃ (mV)</th>
<th>g₁ (mV/g)</th>
<th>g₂ (mV/g)</th>
<th>g₃ (mV/g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Åby River 12</td>
<td>0.0020088</td>
<td>-0.001475</td>
<td>-0.013512</td>
<td>-131.758</td>
<td>-51.332</td>
<td>-83.504</td>
<td>1243</td>
<td>1252</td>
<td>1228</td>
</tr>
<tr>
<td>Åby River 13</td>
<td>-0.0039414</td>
<td>-0.0014096</td>
<td>-0.012954</td>
<td>-146.6064</td>
<td>-66.8836</td>
<td>-105.3206</td>
<td>1250.7739</td>
<td>1260.9438</td>
<td>1235.5634</td>
</tr>
<tr>
<td>Kiruna 14a</td>
<td>-0.00270</td>
<td>-0.00254</td>
<td>-0.01402</td>
<td>-139.98947</td>
<td>-139.97795</td>
<td>-124.85209</td>
<td>1244.9089</td>
<td>1255.7959</td>
<td>1244.42918</td>
</tr>
<tr>
<td>Kiruna 14b</td>
<td>-0.00411</td>
<td>-0.00299</td>
<td>-0.01293</td>
<td>-136.3424</td>
<td>-136.3424</td>
<td>-124.85209</td>
<td>1246.19534</td>
<td>1256.93921</td>
<td>1246.48931</td>
</tr>
<tr>
<td>Kiruna 14c</td>
<td>-0.00281</td>
<td>-0.00104</td>
<td>-0.01303</td>
<td>-138.58103</td>
<td>-138.58103</td>
<td>-124.85209</td>
<td>1248.34494</td>
<td>1259.13240</td>
<td>1248.19218</td>
</tr>
</tbody>
</table>

Table H.6: Calibration parameters for accelerometer 6.

<table>
<thead>
<tr>
<th>Table values</th>
<th>α₁ (rad)</th>
<th>α₂ (rad)</th>
<th>α₃ (rad)</th>
<th>b₁ (mV)</th>
<th>b₂ (mV)</th>
<th>b₃ (mV)</th>
<th>g₁ (mV/g)</th>
<th>g₂ (mV/g)</th>
<th>g₃ (mV/g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Åby River 12</td>
<td>-0.0039414</td>
<td>-0.0014096</td>
<td>-0.012954</td>
<td>-131.758</td>
<td>-51.332</td>
<td>-83.504</td>
<td>1243</td>
<td>1252</td>
<td>1228</td>
</tr>
<tr>
<td>Åby River 13</td>
<td>-0.0039414</td>
<td>-0.0014096</td>
<td>-0.012954</td>
<td>-131.758</td>
<td>-51.332</td>
<td>-83.504</td>
<td>1243</td>
<td>1252</td>
<td>1228</td>
</tr>
<tr>
<td>Kiruna 14a</td>
<td>-0.00411</td>
<td>-0.00299</td>
<td>-0.01293</td>
<td>-136.3424</td>
<td>-136.3424</td>
<td>-124.85209</td>
<td>1244.9089</td>
<td>1255.7959</td>
<td>1244.42918</td>
</tr>
<tr>
<td>Kiruna 14b</td>
<td>-0.00411</td>
<td>-0.00299</td>
<td>-0.01293</td>
<td>-136.3424</td>
<td>-136.3424</td>
<td>-124.85209</td>
<td>1244.9089</td>
<td>1255.7959</td>
<td>1244.42918</td>
</tr>
<tr>
<td>Kiruna 14c</td>
<td>-0.00411</td>
<td>-0.00299</td>
<td>-0.01293</td>
<td>-136.3424</td>
<td>-136.3424</td>
<td>-124.85209</td>
<td>1244.9089</td>
<td>1255.7959</td>
<td>1244.42918</td>
</tr>
</tbody>
</table>
# I Implemented FEM updating methods

This appendix is a brief overview of some of the the main theory on which the results in the previous appendices are based.

## Notation

Table [I.1] gives a summary of the notation used in this appendix. We write $\phi_m^{\text{mea}}$ for the $m$th mode shape obtained from measurements, and $\nu_m^{\text{mea}}$ for its oscillation frequency. The corresponding analytical mode shape and frequency are denoted $\phi_m^a$ and $\nu_m^a$, respectively.

We use boldface characters for vectors and roman characters for matrices. For example, $M$ is the mass matrix, whereas $M$ is the number of mode shapes used for FEM updating.

### I.1 Parametrization

Suppose, for example, that we are interested in the time-dependent displacements in three orthogonal directions of $K$ different points in an analysed structure. Thus, for $D = 3K$ degrees of freedom (DOF), we want to know the displacements $\mathbf{u} = (u_1(t) \ u_2(t) \ \cdots \ u_D(t))^T$ at time $t$. Let $\mathbf{f} = (f_1(t) \ f_2(t) \ \cdots \ f_D(t))^T$ be the external forces applied at the same DOFs. With notation $\dot{\mathbf{u}} = \frac{d\mathbf{u}}{dt}$ and $\ddot{\mathbf{u}} = \frac{d^2\mathbf{u}}{dt^2}$ for derivatives, a discrete linear time-invariant model of structural motion used for finite element models is the second order differential equation

$$ M\ddot{\mathbf{u}} + C\dot{\mathbf{u}} + K\mathbf{u} = \mathbf{f}, \tag{I.1} $$

where $M$, $C$ and $K$ are the $D \times D$ mass, damping and stiffness matrices, respectively.

Ambient vibration measurements, such as those described in appendices C, D, E and F, are usually modeled with finite element models (FEM) for undamped free vibrations, which corresponds to $C = 0$ and $\mathbf{f} = \mathbf{0}$ in (I.1). This is also a convenient model for the forced vibration measurements in Appendix B. The harmonic solutions $\mathbf{u}(t) = \phi_m e^{i2\pi \nu_m t}$ to (I.1) are then obtained from the generalized eigenvalue problem

$$ (-2i\pi \nu_m)^2 M + K) \phi_m = 0 \tag{I.2} $$

### Table I.1: Notation for vectors and index sets used throughout this report and in the MATLAB implementation.

If the measured and predicted mode shapes are compared in $N$ mode points, each with $D_0$ degrees of freedom (DOF), then the total number of DOFs is $D = ND_0$.

<table>
<thead>
<tr>
<th>$\mathbf{a}$</th>
<th>$\mathbf{d} = (d_1, \ldots, D)$</th>
<th>$\mathbf{m} = (m_1, \ldots, M)$</th>
<th>$\mathbf{p} = (p_1, \ldots, P)$</th>
<th>$\phi_m^{\text{mea}} = (\phi_{m,1}^{\text{mea}}, \ldots, \phi_{m,D}^{\text{mea}})^T$</th>
<th>$\phi_m^a = (\phi_{m,1}^a, \ldots, \phi_{m,D}^a)^T$</th>
<th>$\nu_m^{\text{mea}}$</th>
<th>$\nu_m^a$</th>
<th>$\mathbf{v}^T$</th>
<th>$|\mathbf{v}|_2 \equiv \sqrt{\mathbf{v}^T \mathbf{v}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbf{a}$</td>
<td>$d$ is the total number of DOFs.</td>
<td>$M$ is the number of modes to compare.</td>
<td>$P$ is the number of updating parameters.</td>
<td>The $m$th measured mode shape.</td>
<td>The $m$th predicted mode shape.</td>
<td>The $m$th measured mode frequency.</td>
<td>The $m$th predicted mode frequency.</td>
<td>The transpose of the vector $\mathbf{v}$.</td>
<td>The $l^2$-norm of the vector $\mathbf{v}$.</td>
</tr>
</tbody>
</table>
with eigenvectors (or mode shapes) $\phi_m$, eigenvalues
\[ \lambda_m = (2\pi \nu_m)^2 \] (I.3)
and eigenfrequencies $\nu_m$. In the following, we will have both mode shapes $\phi_m^{\text{mea}}$ computed from measurements and analytical mode shapes $\phi_m^a$, computed with FEM software from parameters $a$ in the following way.

It is common practice in structural damage identification to model damages by reduction of the bending stiffness in the damaged part of the structure. Since the bending stiffness $EI$ is proportional to the elasticity modulus $E$, this is suitable, for instance, for simple beam models.

In the case when also torsional components of mode shapes are involved in the measurement data, it is even more advantageous to describe damage by reduction in both bending stiffness $EI$ and torsional stiffness $GI$. In this more general case, the stiffness matrix in the finite element (FE) model can be modeled as
\[ K \overset{\text{def}}{=} K^a = K_u + \sum_{p=1}^P K_p^E (1 - a_p^E) + K_p^G (1 - a_p^G), \quad a_p^E = \frac{E_p^0 - E_p}{E_p^0}, \quad a_p^G = \frac{G_p^0 - G_p}{G_p^0}, \] (I.4)
with $K_u$ being the sum of all unchanged local stiffness matrices. Here $K_p^E$ and $K_p^G$ are the nonzero parts of the element or group constant matrix $K_p$ connected to the degrees of freedom responsible for the bending stiffness and for the torsional stiffness, respectively. (For example, $K_p^E$ is connected to $\{u_x, u_y, u_z, \text{rot}_x, \text{rot}_z\}$ and $K_p^G$ is connected to $\{\text{rot}_y\}$ for the Längforsen bridge in Appendix B).

In this report, we only use the parametrization (I.4) with all $a_p^G = 1$, that is, we only update the elasticity modulus:
\[ K^a = K_u + \sum_{p=1}^P K_p (1 - a_p), \quad a_p \overset{\text{def}}{=} \frac{E_p^0 - E_p}{E_p^0}. \] (I.5)

We noted in Appendix B however that updating of torsion stiffness may be needed for better results using also mode shapes of torsion type. This is therefore an interesting topic and we are currently investigating the possibility to also implement updating of shear modulus.

### I.1.1 Different choices of residual

The goal of the FEM updating method is to choose the parameters $a$ for minimization of certain frequency and mode shape residuals $r_m^a$ and $s_m^a$. The frequency residual is typically a vector with entries
\[ (r_m^a)_{m,d} = \frac{\lambda_m^a - \lambda_m^{\text{mea}}}{\lambda_m^{\text{mea}}} = \frac{(2\pi \nu_m^a)^2 - (2\pi \nu_m^{\text{mea}})^2}{(2\pi \nu_m^{\text{mea}})^2} = \frac{(\nu_m^a)^2 - \nu_m^{\text{mea}}^2}{\nu_m^{\text{mea}}^2}, \quad m = 1, \ldots, M. \] (I.6)

Let $d_m^{\text{max}}$ be the index of an element in $\phi_m^{\text{mea}}$ with maximal absolute value, that is,
\[ |\phi_{m,d_m^{\text{max}}}^{\text{mea}}| \geq |\phi_{m,d}^{\text{mea}}|, \quad \text{for } d = 1, 2, \ldots, D. \] (I.7)

Then one way to define the mode shape residual is the column vector $s_m^a = r_m^{a^\text{max}}$ with
\[ (r_m^{a^\text{max}})_{(m-1)D+d} = \frac{\phi_{m,d}^{a^\text{max}}}{\phi_{m,d}^{a^\text{max}}}, \quad m = 1, \ldots, M, \quad d = 1, \ldots, D. \] (I.8a)

\footnote{Alternatively, (I.3) can also be rewritten in the form $K^a = K_0 - \sum_{p=1}^P (K_p^E a_p^E + K_p^G a_p^G)$, where $K_0$ is the initial value of $K$, corresponding to $a = 0$.}
The normalization is necessary for comparing the shapes. This definition is the same as in \cite{TMDR02, BRDR08}. The normalization to value 1 at DOF $d = \phi_m^{\text{max}}$ is pretty much the same as normalizing $\phi_{m,d}^a$ and $\phi_{m,d}^{\text{mea}}$ to $l^\infty$-norm 1, which could make this method a bit sensitive to measurement errors at DOF $d = \phi_m^{\text{max}}$. Therefore, we have also tried normalizing to $l^2$-norm 1 instead. With a constant $c$ to guard against “opposite signs” in the compared mode shapes, we then define the residual $r_m^a = r_m^{\text{res}}$ with

$$(r_m^{\text{res}})_{(m-1)D+d} \overset{\text{def}}{=} \frac{1}{\|\phi_m^a\|_2} \phi_m^a - \frac{c}{\|\phi_m^{\text{mea}}\|_2} \phi_m^{\text{mea}}, \quad c = \frac{\text{sgn}(\phi_m^{\text{mea}})}{\text{sgn}(\phi_m^{\text{mea}})} \quad m = 1, \ldots, M. \quad (I.8b)$$

Here $\text{sgn}$ is the sign function $\text{sgn}(x) \overset{\text{def}}{=} \frac{|x|}{x}$. This residual also is a bit more “democratic” in the sense that that the contribution from each mode shape is of size less than two, and thus of the same order of magnitude as the relative error contribution from each frequency in $r_i^a$. This should be a good basis for potential further adjustments via the weights in the weight matrix $W_r$ in \cite{I10} below. With notation $P_a u$ for the orthogonal projection of the vector $u$ on the vector $v$, one other option is to make an orthogonal decomposition of $\phi_m^a$ into $\phi_m^a = P_{\phi_m^{\text{mea}}} \phi_m^a + (\phi_m^a - P_{\phi_m^{\text{mea}}} \phi_m^a) \overset{\text{def}}{=} P_{\phi_m^{\text{mea}}} \phi_m^a + r_m^{\text{orth},m}$, where the first term is proportional to the measured mode shape and the second term is the orthogonal error term or residual (see Figure \ref{fig:MSF_residuals}). Thus, with notation $\hat{v} \overset{\text{def}}{=} \frac{1}{\|v\|_2} v$ for unit vectors, we define $r_m^a = r_m^{\text{orth}} = (r_m^{\text{orth},1}^a, r_m^{\text{orth},2}^a, \ldots, r_m^{\text{orth},M}^a)^T$ with

$$r_m^{\text{orth},m} \overset{\text{def}}{=} \frac{1}{\phi_m^{\text{mea}}} (\phi_m^a - P_{\phi_m^{\text{mea}}} \phi_m^a) = \frac{1}{\phi_m^{\text{mea}}} (\phi_m^a - (\phi_m^{\text{mea}}^T \phi_m^a) \phi_m^{\text{mea}}). \quad (I.8c)$$

The division with $\phi_m^{\text{mea}}$ gives a dimensionless relative error. With the roles of $\phi_m^{\text{mea}}$ and $\phi_m^a$ interchanged\footnote{The roles of $\phi_m^{\text{mea}}$ and $\phi_m^a$ were interchanged because we regard the analytical mode shapes $\phi_m^a$ as approximations of the measured mode shapes $\phi_m^{\text{mea}}$. For joint minimization of exactly this error for all $m$, we get the residual \cite{I8c} at the cost of a more complicated differentiation in \cite{I16c}.}, this residual was used in \cite{RTDR10} with notation $\text{MSF}(u, v) = \frac{u^T v}{\|u\|^2}$ for the modal scale factor introduced in \cite{AB82}.

**Remark 2.** In the current version of the MATLAB implementation, we have skipped the normalization factor $1/\phi_m^{\text{mea}}$ in \cite{I8c} with obvious simplification of the differentiation in \cite{I16c}.

![Figure I.1](image-url) 

**Figure I.1:** The MSF residuals based on an orthogonal decomposition of $\phi_m^a$ into one vector parallel to $\phi_m^{\text{mea}}$ and one residual vector orthogonal to $\phi_m^{\text{mea}}$. 
This gives a default weighting of the $m$ different errors that is proportional to how the different mode shapes are scaled by the FEM software, usually normalization so that $\phi_m^a T M \phi_m^a = 1$, with $M$ being the mass matrix in I.1. One further alternative is to do scaling to $L^2$-norm 1 in (I.8c), for the reasons explained after (I.8b).

One more possible choice is to define the column vector residual $r_s^a = \rho_{\sin^2}$ with
\[
(r_{\sin^2})_m \equiv 1 - \frac{(\phi_m^a T \phi_m^{\text{mea}})^2}{\phi_m^a T \phi_m^{\text{mea}} T \phi_m^{\text{mea}}} = \sin^2(\theta_m), \quad m = 1, 2, \ldots, M. \tag{I.8d}
\]
with $\theta_m$ being the angle between the vectors $\phi_m^a$ and $\phi_m^{\text{mea}}$. This could be a reasonable residual for some applications, but it reduces the total residual $r(a)$ in (I.10) to a vector of length $2M$, which is less than the number of parameters in $a$ for our applications. Hence, (I.8d) reduces the measurements to fewer observations than unknown parameters, for which no unique solution can be expected for the minimization problem described below. Thus we do not use (I.8d) for any results in this report. For more mode shape residuals, see, for example, [Jai05, Section 3.2].

We can now define the residual
\[
r \equiv r(a) \equiv \left( \begin{array}{c}
\rho_{\sin^2} \\
\end{array} \right) = W_r \left( \begin{array}{c}
\rho_{\sin^2} \\
\end{array} \right), \quad R = M + MD = M(D + 1). \tag{I.9}
\]
The diagonal weight matrix $W_r$ allows to, for example, focus on mode shapes and frequencies with good signal-to-noise ratio.

### I.1.2 The objective function and its Taylor approximation

Our goal is to choose $a$ for minimizing the objective function
\[
f(a) \equiv \frac{1}{2} r(a)^T W r(a) = \frac{1}{2} \sum_{\rho=1}^{R} w_{\rho}^2 \rho_{\rho}(a)^2, \quad W \equiv W_r^T W_r = \left( \begin{array}{c}
w_1^2 \\
\vdots \\
w_R^2 \\
\end{array} \right).
\tag{I.10}
\]
The trust region approach used for for minimizing $f$ is based on the Taylor approximation
\[
f(a_0 + h) \approx f(a_0) + (\nabla f)(a_0)^T h + \frac{1}{2} h^T H_f(a_0) h \equiv m(a). \tag{I.11}
\]
for which the gradient $\nabla f$ and the Hessian matrix $H_f$ can be computed from (I.10) as follows. The gradient is the $P \times 1$ vector with $p$th entry
\[
(\nabla f)_p = \frac{\partial f}{\partial a_p} = \sum_{\rho=1}^{R} w_{\rho}^2 \rho_{\rho}(a) \frac{\partial \rho_{\rho}(a)}{\partial a_p},
\]
\[
\nabla f = \sum_{\rho=1}^{R} w_{\rho}^2 \rho_{\rho}(a)(\nabla \rho_{\rho}(a)) = J_r^T(a) W r(a), \quad \text{for the Jacobian } (J_r(a))_{\rho,p} \equiv \frac{\partial \rho_{\rho}(a)}{\partial a_p} \tag{I.12}
\]
Similarly, for the Hessian

\[
(H_f)_{k,l} \overset{\text{def}}{=} \frac{\partial^2 f}{\partial a_k \partial a_l} = \frac{\partial^2}{\partial a_k \partial a_l} \left( w_1^2 r_1(a)^2 + \cdots + w_R^2 r_R(a)^2 \right) = \frac{\partial}{\partial a_l} \sum_{\rho=1}^{R} w_{\rho}^2 r_{\rho}(a) \frac{\partial r_{\rho}(a)}{\partial a_l}
\]

\[
= \sum_{\rho=1}^{R} \left( \frac{\partial w_{\rho}^2 r_{\rho}(a)}{\partial a_k} \frac{\partial w_{\rho} r_{\rho}(a)}{\partial a_l} + w_{\rho}^2 r_{\rho}(a) \frac{\partial^2 r_{\rho}(a)}{\partial a_k \partial a_l} \right)
\]

\[
= \sum_{\rho=1}^{R} \left( \frac{\partial w_{\rho}^2 r_{\rho}(a)}{\partial a_k} \frac{\partial w_{\rho} r_{\rho}(a)}{\partial a_l} + w_{\rho}^2 r_{\rho}(a) (H_r(a))_{k,l} \right) .
\]

(1.13)

\[H_f = J_r(a)^T W J_r(a) + \sum_{\rho=1}^{R} w_{\rho}^2 r_{\rho}(a) H_r(a) \approx J_r(a)^T W J_r(a).
\]

(1.14)

The last approximation is reasonable since for the applications in this report, we start from small \(r_{\rho}(a)\) in the first iteration and then \(r_{\rho}(a)\) can be expected to get smaller for each iteration. Since \(r_{\infty}^a\) is a length \(M\) vector and \(r_{\rho}^a\) has length \(MD\), the residual \(r\) defined in (1.9) has a \(M(D + 1) \times P\) Jacobian \(J_r = \begin{bmatrix} J_r^a & 0 \end{bmatrix}\). From (1.6) with predicted eigenvalues \(\lambda_m^a = (2\pi \nu_m^a)^2\) as in (1.3), we get

\[(J_r^a)_{m,p} \overset{\text{def}}{=} \frac{\partial (r_{\infty}^a)_m}{\partial a_p} = \frac{1}{(\lambda_m^a)^2} \frac{\partial (2\pi \nu_m^FEM)^2}{\partial a_p}, \quad m = 1, \ldots, M \quad (1.15)
\]

The \(MD \times P\) Jacobian \(J_r^a\) depends on the definition of \(r_{\infty}^a\). For the residual \(r_{\infty}^a = r_{\infty}^a\) defined in (1.8a),

\[(J_r^a)_{(m-1)D+d,p} \overset{\text{def}}{=} \frac{\partial (r_{\infty}^a)_{(m-1)D+d}}{\partial a_p} = \frac{\partial \phi_{m,d}^a}{\partial a_p} \frac{\phi_{m,d}^a}{\phi_{m,d}^a} - \frac{\phi_{m,d}^a}{\phi_{m,d}^a} \frac{\partial \phi_{m,d}^a}{\partial a_p}, \quad m = 1, \ldots, M, \quad d = 1, \ldots, D .
\]

(1.16a)

For the \(l^2\)-normalized residual \(r_{\infty}^a = r_{\infty}^a\) defined in (1.8b),

\[(J_r^a)_{(m-1)D+d,p} \overset{\text{def}}{=} \frac{\partial (r_{\infty}^a)_{(m-1)D+d}}{\partial a_p} = \frac{\partial \phi_{m,d}^a}{\partial a_p} \frac{\phi_{m,d}^a}{\phi_{m,d}^a} - \frac{\phi_{m,d}^a}{\phi_{m,d}^a} \frac{\partial \phi_{m,d}^a}{\partial a_p} = \frac{1}{2} \frac{\partial (\phi_{m,d}^a)^2}{\partial a_p} \frac{\phi_{m,d}^a}{\phi_{m,d}^a} .
\]

(1.16b)

Next, recall from (1.16a) that

\[
\frac{\partial}{\partial a_p} \frac{1}{\phi_{m,d}^a} \phi_{m,d}^a = \frac{1}{\phi_{m,d}^a} \left( \frac{\partial \phi_{m,d}^a}{\partial a_p} - \frac{\partial \phi_{m,d}^a}{\partial a_p} \right).
\]

Hence, for \(r_{\infty}^a = r_{\infty}^a\) and with index set \(I_m \overset{\text{def}}{=} ((m-1)D+1, (m-1)D+2, \ldots, (m-1)D+D)\), we get from (1.8c) that
\[(J_{r_{\text{orth}}}^{a})_{m,p} = \frac{\partial (r_{\text{orth}}^{a})_{m}}{\partial a_{p}} = \frac{\partial}{\partial a_{p}} \left( \frac{1}{\phi_{m,d_{\text{max}}}} \phi_{m}^{a} - \frac{1}{\phi_{m,d_{\text{max}}}} \bar{\phi}_{m}^{a} \phi_{m} \right) \]
\[
= \frac{1}{\phi_{m,d_{\text{max}}}} \left[ \phi_{m,d_{\text{max}}}^{a} \frac{\partial \phi_{m}}{\partial a_{p}} - \phi_{m}^{a} \frac{\partial \phi_{m,d_{\text{max}}}}{\partial a_{p}} \right] - \left( \bar{\phi}_{m}^{a} \phi_{m}^{a} \phi_{m,d_{\text{max}}}^{a} - \phi_{m}^{a} \phi_{m,d_{\text{max}}}^{a} \phi_{m} \right) \bar{\phi}_{m}, \quad (I.16c)
\]
or equivalently
\[(J_{r_{\text{orth}}}^{a})_{m,p} = (J_{r_{\text{orth}}}^{a})_{m,p} - \left( \bar{\phi}_{m}^{a} \phi_{m} \right) \]

Similarly, it follows for \(r_{s}^{a} = r_{s}^{a,2}\) defined by \(I.8d\) that
\[(J_{r_{s}^{a,2}}^{a})_{m,p} \overset{\text{def}}{=} \frac{\partial (r_{s}^{a})_{m}}{\partial a_{p}} = - \frac{\partial (\phi_{m}^{a} \phi_{m}^{a})^{2}}{\phi_{m}^{a} \phi_{m}} \phi_{m}^{a} - \frac{\partial (\phi_{m,d_{\text{max}}}^{a} \phi_{m})^{2}}{\phi_{m,d_{\text{max}}}^{a} \phi_{m}} \phi_{m}^{a} - (\phi_{m}^{a} \phi_{m,d_{\text{max}}}^{a} \phi_{m})^{2} \frac{\partial \phi_{m}^{a} \phi_{m}^{a}}{\partial a_{p}}
\]
\[
= - 2 (\phi_{m}^{a} \phi_{m,d_{\text{max}}}^{a} \phi_{m})^{2} \frac{\partial \phi_{m}^{a} \phi_{m}^{a}}{\partial a_{p}}
\]
\[
= - 2 \frac{\phi_{m}^{a} \phi_{m,d_{\text{max}}}^{a} \phi_{m}^{a}}{\parallel \phi_{m}^{a} \parallel_{2}^{2}} \phi_{m}^{a} \phi_{m}^{a} \phi_{m,d_{\text{max}}}^{a} \phi_{m}^{a} - \left( \phi_{m}^{a} \phi_{m,d_{\text{max}}}^{a} \phi_{m}^{a} \right) \phi_{m}^{a}, \quad (I.16d)
\]

with \(I\) being the identity matrix.

These expressions depend on the derivatives \(\frac{\partial (2\pi r_{\text{EM}}^{a})^{2}}{\partial a_{p}}\) and \(\frac{\partial \phi_{m}^{a}}{\partial a_{p}}\) which we can compute, as in [TMDR02], from, for instance the Fox–Kapoor formulae \(I.5\) below.

### I.2 Derivatives of eigenvalues

Consider the generalized eigenvalue problem \(I.2\) written in the form
\[(K - \lambda_{m} M+) \phi_{m} = 0,
\]

with stiffness matrix \(K\) and mass matrix \(M\) that in structural mechanics both are positive semidefinite. The eigenvectors \(\phi_{m}\) are usually normalized so that
\[\phi_{m}^{T} M \phi_{m} = \delta_{m,n},
\]

(see, for instance, [CMPW01], Appendix C]).

For the parametrization \(K = K^{a} = K_{u} + \sum_{p=1}^{P} K_{p}(1 - a_{p})\) in \(I.5\), the eigenvectors \(\phi_{m}\), the eigenvalues \(\lambda_{m}\) and \(K\) will depend on the parameters \(a_{p}\), but the mass matrix \(M\) does not depend on \(a\). We will now present two different methods for computing the derivatives of eigenvalues, depending on whether there are repeated eigenvalues or not.
Fox–Kapoor for non-repeated eigenvalues  For non-repeated eigenvalues, Fox and Kapoor derived equations \cite[eqs. (12),(18),(20)]{FK68} for the derivatives of eigenvalues and eigenvectors with respect to the parameters \(a_p\). For the parametrization \((I.5)\), only the stiffness matrix \(K^a\) is affected by the parameters \(a_p\). Then the Fox–Kapoor formulae are

\[
\frac{\partial \lambda_m^a}{\partial a_p} = \phi_m^a \, \frac{\partial K^a}{\partial a_p} \phi_m^a = -\phi_m^a \,^T K_p \phi_m^a, \quad (I.17a)
\]

\[
\frac{\partial \phi_m^a}{\partial a_p} = \sum_{q \neq m} \frac{\phi_q^a \,^T K_p \phi_m^a}{(\lambda_m^a)^2 - (\lambda_q^a)^2} \phi_q^a = \sum_{q \neq m} \frac{\phi_q^a \,^T K_p \phi_m^a}{(\lambda_q^a)^2 - (\lambda_m^a)^2} \phi_q^a, \quad (I.17b)
\]

with predicted eigenvalues \(\lambda_m^a = (2 \pi \nu_m^a)^2\) as in \((I.3)\). Previous version of the FEM updating software described in Appendix \(J\) used these formulae, which have two main disadvantages: Equation \((I.17b)\) does not hold for repeated eigenvalues and the sum in \((I.17b)\) usually needs to be truncated for efficient computation, which introduces a truncation error of unknown size. The Kirchhoff plate in Appendix \(G\) has several repeated eigenvalues. Towers (such as the one in Appendix \(F\)) are other structures that for symmetry reasons can be expected to have repeated eigenvalues. Therefore, we took care of both the mentioned disadvantages by implementing Nelson’s method.

Nelson’s method  The following theorem summarizes the simplest version of Nelson’s method, which works also for repeated eigenvalues and without any truncation or other approximations. We refer to \cite{Fri96} for a generalization that also covers the case of coinciding derivatives \(\frac{\partial \lambda_k}{\partial a_p} \neq \frac{\partial \lambda_m}{\partial a_p}\) for some \(k \neq m\).

**Theorem 1.** For symmetric positive semifinite \(D \times D\) matrices \(M\) and \(K \overset{\text{def}}{=} K^a\) with parametrization \((I.5)\), suppose for a fixed value of the parameter vector \(a\) that the eigenvalue problem

\[
(K - \lambda M)\phi_m = 0 \quad \text{with} \quad \phi_m^T M \phi_m = \delta_{k,m}, \quad (I.18)
\]

has an eigenvalue \(\lambda_m = \lambda_m^a\) of multiplicity \(M = M_a\). Store \(M\) corresponding orthogonal eigenvectors \(\phi_m = \phi_m^a\) as column vectors in a matrix \(\Phi = (\phi_1, \phi_2, \cdots, \phi_M)\). Store the eigenvectors of \(-\Phi^T K_p \Phi\) as column vectors in a matrix \(H\). Then, the column vectors of

\[
\Psi \overset{\text{def}}{=} \Psi^a \overset{\text{def}}{=} \Phi H \overset{\text{def}}{=} (\psi_1, \psi_2, \cdots, \psi_M)
\]

are also solutions of \((I.18)\) with the same eigenvalue \(\lambda_1 = \lambda_2 = \cdots = \lambda_M\). Moreover, for each \(K_p\) in the parametrisation \((I.5)\),

\[
-\Psi^T K_p \Psi = \begin{pmatrix}
\frac{\partial \lambda_1}{\partial a_p} & 0 & \cdots & 0 \\
0 & \frac{\partial \lambda_2}{\partial a_p} & \cdots & 0 \\
\vdots & \ddots & \ddots & \vdots \\
0 & \cdots & 0 & \frac{\partial \lambda_M}{\partial a_p}
\end{pmatrix}.
\quad (I.19)
\]

Suppose that \(\frac{\partial \lambda_k}{\partial a_p} \neq \frac{\partial \lambda_m}{\partial a_p}\) for \(k \neq m\) in \((I.19)\) and that \(K - \lambda M\) becomes a full rank matrix for a certain choice of removed rows and columns, say columns and rows number \(k_1, k_2, \ldots, k_M\). (See suggestions for how to make this choice in Remark \(3\) below). Then

\[
\frac{\partial \psi_m}{\partial a_p} = v_{m,p} + \Psi c_{m,p} \quad (I.20)
\]
With \( \mathbf{v}_{m,p} \) uniquely determined from

\[
(K - \lambda_m \mathbf{M}) \mathbf{v}_{m,p} = \left( K_p + \frac{\partial \lambda_m}{\partial a_p} \mathbf{M} \right) \mathbf{\psi}_m = \mathbf{b}_{m,p}
\]  

(1.21)

after setting elements number \( k_1, k_2, \ldots, k_M \) in \( \mathbf{b}_{m,p} \) to zero and replacing rows and columns number \( k_1, k_2, \ldots, k_M \) in \( K = \lambda_m \mathbf{M} \) with the corresponding rows and columns of the \( D \times D \) identity matrix. Finally, the coefficient vector \( \mathbf{c}_{m,p} \) in (1.20) is the \( m \)th column of the matrix \( \mathbf{C}_p \) given by

\[
[C_p]_{m,m} = -\mathbf{v}_{m,p}^T \mathbf{M} \mathbf{\psi}_m, \quad [C_p]_{k,m} = \frac{\mathbf{v}_k^T \left( K_p + \frac{\partial \lambda_m}{\partial a_p} \mathbf{M} \right) \mathbf{v}_{m,p}}{\left( \frac{\partial \lambda_k}{\partial a_p} - \frac{\partial \lambda_m}{\partial a_p} \right)}, \quad k, m = 1, \ldots, M, \quad k \neq m.
\]

(1.22)

Proof. First, let \( \mathbf{H} = [\mathbf{h}_1 \ \mathbf{h}_2 \ \cdots \ \mathbf{h}_M] \) be an arbitrary orthogonal \( \mathcal{M} \times \mathcal{M} \)-matrix (that is, \( \mathbf{H}^T \mathbf{H} = \mathbf{I} \)). Then the column vectors of \( \mathbf{\Psi} \) are also solutions of (1.18) with the same eigenvalue. Indeed, \( \mathbf{\psi}_k^T \mathbf{M} \mathbf{\psi}_m = \mathbf{h}_k^T \mathbf{\Phi}^T \mathbf{M} \mathbf{h}_m = \mathbf{h}_k^T \mathbf{h}_m = \delta_{k,m} \) and it follows from (1.18) that

\[
(K - \lambda_m \mathbf{M}) \mathbf{h}_m = 0 \rightarrow \mathbf{h}_m = 0.
\]

(1.23)

For other parameter values \( \mathbf{a} + \Delta \mathbf{a} \), the eigenvalues are in general no longer identical, so differentiation of (1.23) with respect to \( a_p \) adds restrictions on \( \mathbf{H} \) that are necessary for differentiability of the eigenvectors. Differentiation of (1.23) and insertion of (1.5) and (1.18) gives that

\[
0 = \frac{\partial}{\partial a_p} \left( K - \lambda_m \mathbf{M} \right) \mathbf{h}_m = \left( \frac{\partial \mathbf{K}}{\partial a_p} - \frac{\partial \lambda_m}{\partial a_p} \mathbf{M} - \lambda_m \frac{\partial \mathbf{M}}{\partial a_p} \right) \mathbf{h}_m + (K - \lambda_m \mathbf{M}) \left( \frac{\partial \mathbf{\Phi}}{\partial a_p} \mathbf{h}_m + \frac{\partial \mathbf{h}_m}{\partial a_p} \right)
\]

\[
0 = \left( -K_p - \frac{\partial \lambda_m}{\partial a_p} \mathbf{M} \right) \mathbf{h}_m + (K - \lambda_m \mathbf{M}) \frac{\partial \mathbf{\Phi}}{\partial a_p} \mathbf{h}_m.
\]

(1.24)

Multiplication from the left with \( \mathbf{\Phi}^T \) and using the symmetry of \( \mathbf{M} \) and \( \mathbf{K} \) combined with (1.18) gives that

\[
0 = \left( -\mathbf{\Phi}^T K_p \mathbf{\Phi} - \frac{\partial \lambda_m}{\partial a_p} \mathbf{\Phi}^T \mathbf{M} \mathbf{\Phi} \right) \mathbf{h}_m + ((K - \lambda_m \mathbf{M}) \mathbf{\Phi})^T \frac{\partial \mathbf{\Phi}}{\partial a_p} \mathbf{h}_m
\]

\[
(-\mathbf{\Phi}^T K_p \mathbf{\Phi}) \mathbf{h}_m = \frac{\partial \lambda_m}{\partial a_p} \mathbf{h}_m.
\]

(1.25)

Thus the column vectors of \( \mathbf{H} \) are eigenvectors of \( -\mathbf{\Phi}^T K_p \mathbf{\Phi} \) with corresponding eigenvalues \( \frac{\partial \lambda_m}{\partial a_p} \), and (1.19) now follows from (1.25).

Here the derivative \( \frac{\partial \lambda_k}{\partial a_p} \) corresponds to the eigenvector \( \mathbf{\psi}_m \) for which we want to write the derivative in the form (1.20). The second term is needed because in the following approach, not all elements of \( \mathbf{v}_{m,p} \) can be computed. It follows directly from (1.24) that

\[
0 = \left( -K_p - \frac{\partial \lambda_m}{\partial a_p} \mathbf{M} \right) \mathbf{\psi}_m + (K - \lambda_m \mathbf{M}) \frac{\partial \mathbf{\psi}_m}{\partial a_p}.
\]

(1.26)

Insertion of (1.20) and (1.23) gives (1.21). The matrix \( K - \lambda_m \mathbf{M} \) is rank deficient with rank \( D - \mathcal{M}_a \) [Str05, Section 2.4]. Therefore we reduce the number of equations to \( D - \mathcal{M}_a \) in (1.21) by setting elements number \( k_1, k_2, \ldots, k_M \) in \( \mathbf{b}_{m,p} \) to zero and by replacing rows and columns
number $k_1,k_2,\dots,k_M$ in $K - \lambda_m M$ with the corresponding rows and columns of the $D \times D$ identity matrix. (This means that we remove equations $k_1,k_2,\ldots,k_M$ and set the corresponding elements in $v_{m,p}$ to zero.)

To determine all elements of $\frac{\partial \phi_m}{\partial a_p}$, it remains to compute the coefficient vectors $c_{m,p}$ in (I.20). Organize them as the column vectors of the matrix $C_p$, and let the vectors $v_{m,p}$ be the corresponding columns of a matrix $V_p$. From (I.18) it follows that $\Psi^T M \Psi = I_{M \times M}$. For $M$ not depending on the parameters $a_p$, differentiation gives

$$0 = \frac{\partial \Psi^T}{\partial a_p} M \Psi + \Psi^T M \frac{\partial \Psi}{\partial a_p}$$

(insert (I.20))

$$0 = (V_p + \Psi C_p)^T M \Psi + \Psi^T M (V_p + \Psi C_p)$$

(use that $\Psi^T M \Psi = I_{M \times M}$)

$$C_p^T + C_p = - V_p^T M \Psi - \Psi^T M V_p$$

We cannot assume $C_p$ to be symmetric, so using the symmetry of $M$, this only gives that the $m$th diagonal element of $C_p$ is given by (1.22). For finding the off-diagonal elements $[C_p]_{i,j}$, we use an approach similar to how we found the eigenvalue derivatives. Differentiation of (I.26) with respect to $a_p$ gives

$$0 = - \frac{\partial^2 \lambda_m}{\partial a_p} M \psi_m - 2 \left( K_p + \frac{\partial \lambda_m}{\partial a_p} M \right) \frac{\partial \psi_m}{\partial a_p} + (K - \lambda_m M) \frac{\partial^2 \psi_m}{\partial a_p^2}.$$

By multiplication from the left with $\psi_k^T$ for $k \neq m$, inserting (I.20), (I.23), using the symmetry of $K,M$ and then using that $\psi_k^T M \psi_m = 0$, we get

$$0 = - \psi_k^T \frac{\partial^2 \lambda_m}{\partial a_p^2} M \psi_m - 2 \psi_k^T \left( K_p + \frac{\partial \lambda_m}{\partial a_p} M \right) \left( v_{m,p} + \Psi c_{m,p} \right) + \psi_k^T (K - \lambda_m M) \frac{\partial v_{m,p} + \Psi c_{m,p}}{\partial a_p}$$

$$= - 2 \psi_k^T \left( K_p + \frac{\partial \lambda_m}{\partial a_p} M \right) \left( v_{m,p} + \Psi c_{m,p} \right)$$

$$= - 2 \psi_k^T \left( K_p + \frac{\partial \lambda_m}{\partial a_p} M \right) \psi_{m,p} - 2 \psi_k^T \left( K_p + \frac{\partial \lambda_m}{\partial a_p} M \right) \Psi c_{m,p}.$$

For the last term, note that $\psi_k^T M \Psi c_{m,p} = [C_p]_{k,m}$. Consequently, it follows from (I.19) that $-\psi_k^T K_p \Psi c_{m,p} = \frac{\partial \lambda_m}{\partial a_p} [C_p]_{k,m}$. Thus (I.22) follows.

**Remark 3.** In Theorem 1 and its proof, it remains to decide which $M$ equations to remove from (I.21) (thus setting the corresponding elements of $v_{m,p}$ to zero). We choose the positions corresponding to the element with largest absolute value in each $\psi_m$. More precisely, we take the eigenvectors $\psi_m$ one by one and among the element positions not already chosen, we pick the position of the element with the largest absolute value. See [Fri96] for some alternative approaches and further references.

**Remark 4.** Although there is an infinite number of orthogonal bases for the eigenspace of (I.18) with eigenvalue $\lambda_m$, the main idea of the proof is that, at least for some $a_p$, a small change in the corresponding updating parameter will cause a bifurcation into different eigenvalues and a uniquely determined orthogonal basis. This basis was automatically chosen via the continuity (or rather differentiability) that is required for the implicit differentiation of (I.23) with respect to $a_p$ to make sense. The new eigenbasis $\Psi \overset{\text{def}}{=} \Phi H$ is the one for which derivatives are computed, but in our current implementation, we keep the original basis $\Phi$ when computing the objective function and use the computed derivatives of $\Psi$ as an approximation of the corresponding
derivatives of $\Phi$ (which generally can be expected to be of the same order of magnitude). Thus there is an inherent approximation in the current implementation of the Nelson method. For the simulated plate we kept the eigenvectors $\Phi$ because they matched the corresponding "measured" mode shapes (both computed by the same MATLAB commands). In this case perhaps the computed derivatives should rather be multiplied from the right with $H^{-1}$, but for measurements on a real structure, it is not clear how the chosen method for modal analysis choose one particular set of eigenvectors, say $\Phi_{\text{mea}}$. Nonetheless, we would like the predicted eigenvector basis $\Phi$ to be as similar as possible to $\Phi_{\text{mea}}$, but at the same time, we would also like to replace it with the eigenspace basis $\Psi$ to avoid approximations in the Nelson method, which gives a problem if $\Phi_{\text{mea}}$, $\Phi$ and $\Psi$ are clearly different. Further investigation of this problem is out of the scope of this project, but an important issue for avoiding unnecessary approximation in further development of the MATLAB implementation.

Note in the above proof that for the special case of nonrepeated eigenvalues ($M = 1$), the computed derivatives coincides with (I.17a).

I.3 Regularization with interpolating basis functions

A large number of updating parameters $a_p$ can result in an ill-conditioned Jacobian $J_r$. One countermove suggested in [RTDR10, TMDR02] is to do the FEM updating of $a_p$ only for $p$ in a subsequence $\mathcal{P} = [P_1, P_2, \ldots, P_R]$ of $[1, 2, 3, \ldots, P]$ and then use interpolation for deciding the value of the remaining parameters $a_p$.

For example, consider a 1D-structure that is divided into 22 groups of elements with centre points $x_p$, as illustrated in Figure I.2 (a). The blue circles indicate a coarser grid of points with indices $\mathcal{P} = [1, 4, 7, 10, 13, 16, 19, 22]$. Now let $b_{\mathcal{P}_k}(x)$ be functions with the so-called interpolation property

$$b_{\mathcal{P}_k}(x) = \delta_{r,k} \overset{\text{def}}{=} \begin{cases} 1 & \text{if } r = k, \\ 0 & \text{if } r \neq k. \end{cases}$$ (I.27a)

Then, for

$$a(x) \overset{\text{def}}{=} \sum_{r=1}^{R} a_{\mathcal{P}_r} b_{\mathcal{P}_r}(x) \quad \text{it follows that} \quad a(x_{\mathcal{P}_r}) = a_{\mathcal{P}_r}. \quad (I.27b)$$

For the results in this report We have used piecewise linear interpolating functions $b_k(x)$ that are linear on each interval $[x_{\mathcal{P}_r}, x_{\mathcal{P}_{r+1}}]$, sometimes called tent functions. For instance, see the tent function $b_7(x)$ plotted in Figure I.2 (a).

For 2D-structures (like the concrete plate in Appendix B, we can do a construction of a separable interpolating basis, as suggested in [SD+09]. Then, with notation as in Figure I.2 (b),

Figure I.2: (a) Tent functions for a 1D structure. (b) Tent functions for a 2D structure.
the interpolating basis function centred at the point \( P_0 \) is

\[
b(\xi, \eta) \overset{\text{def}}{=} T_1(\xi)T_2(\eta)
\]

with local coordinates \((\xi, \eta)\) defined as in the figure. There, blue circles indicate six neighbouring points in the coarse grid.

For instance, suppose that we have \( P = 35 \) updating parameters \( a_p \) in points \((x_p, y_p)\), ordered in a grid

\[
a_1 \quad a_8 \quad a_{15} \quad a_{22} \quad a_{29}
\]
\[
a_2 \quad a_9 \quad a_{16} \quad a_{23} \quad a_{30}
\]
\[
a_3 \quad a_{10} \quad a_{17} \quad a_{24} \quad a_{31}
\]
\[
a_4 \quad a_{11} \quad a_{18} \quad a_{25} \quad a_{32}
\]
\[
a_5 \quad a_{12} \quad a_{19} \quad a_{26} \quad a_{33}
\]
\[
a_6 \quad a_{13} \quad a_{20} \quad a_{27} \quad a_{34}
\]
\[
a_7 \quad a_{14} \quad a_{21} \quad a_{28} \quad a_{35}
\]

with blue colour for the parameters of the coarse grid. Now the coarse grid indices are

\[
P = [1, 4, 7, 15, 18, 21, 29, 32, 35] \overset{\text{def}}{=} [P_1, P_2, \ldots, P_9].
\]

This gives \( R = 9 \) tent functions, all with the interpolation property

\[
b_{P_k}(x_{P_r}, y_{P_r}) = \delta_{r,k}
\]

similar to \((I.27a)\).

Again, the FEMU software is allowed to choose the parameters \( a_p \) freely for \( p \in P \), and then all other \( a_p \) are defined as the interpolation

\[
a_p \overset{\text{def}}{=} a(x_p, y_p), \quad \text{for} \quad a(x, y) \overset{\text{def}}{=} \sum_{r=1}^{R} a_{R_r} b_{R_r}(x, y).
\]

Let \( L \) be the \( P \times R \)-matrix

\[
L = \begin{pmatrix} b_{P_1} & b_2 & \cdots & b_{P_R} \end{pmatrix} \quad \text{with} \quad b_r = \begin{pmatrix} b_{r}(x_1, y_1) \\ b_{r}(x_2, y_2) \\ \vdots \\ b_{r}(x_p, y_p) \end{pmatrix}
\]

(computed by the method \texttt{IBFemOptParamConv} in Appendix J.6.5.1, page 198).

If \( a^R \) is the vector of updating paremeters in the coarse grid, then \((I.27e)\) gives that

\[
a = L a^R.
\]

This conversion is performed by the method \texttt{convertOpt2FEM} in Appendix J.6.5.4, page 198.

Moreover, from \((I.29)\) follows that the Jacobian

\[
(J_{r})_{d,p}^R \overset{\text{def}}{=} \frac{\partial}{\partial \alpha_{R_r}} \sum_{p=1}^{P} \frac{\partial \alpha_{p}}{\partial \alpha_{R_r}} = \sum_{p=1}^{P} (J_{r})_{d,p}^R \frac{\partial (L a^R)^p}{\partial \alpha_{R_r}} = \sum_{p=1}^{P} (J_{r})_{d,p}^R L_{p,r}
\]

\[
J_{r}^R = J_{r}^R L
\]
I.3 Regularization with interpolating basis functions

We show in Appendix J.6.7.1, page 208 how this relation is used for computing the Jacobian as well as the corresponding gradient and Hessian.

Figure I.3 shows the resulting basis functions for one example with 9 points in the coarse mesh. This mesh is uniform, and then everything works fine. For perturbed such grid we

Figure I.3: For the mesh in (a), with red colour for the course mesh, the separable basis functions defined in (I.27c) are plotted in (b).
I implemented FEM updating methods get the scenario in Figure 1.4. The interpolation property is now only approximately true. For example, the basis function centered in the point labeled 64 will equal zero in the points labeled 6, 40, 45 and 54, but will only be approximately 0 in the points labeled 38, 83, 84 and 88. One could try to improve this by defining the basis functions as in SD+09 Figure 1, or

![Figure 1.4: The results of perturbed mesh points in Figure 1.3.](image-url)
correspondingly for an irregular mesh shown in Figure I.2(c). For instance, \( b_{ls}(x,y) \) is planar in each of the eight triangles indicated in Figure I.2(c) with the basis function defined to be planar in each of the indicated triangles. This construction was used in [Gr14] because it keeps the interpolation property also for perturbed rectangular meshes. However, it does not give the intended interpolation. For example, it is easy to check that if the coarse grid consists only of the points \((-1,-1), (1,-1), (1,1) \) and \((-1,1)\), then with \( \alpha_p = 1 \) in these four points, the construction (I.27a) gives \( a(0,0) = 2 \), which is not an interpolation of the values in the corner points. The separable basis defined in (I.27c) has faster nonlinear decay along the diagonal, which for the just described example gives \( a(0,0) = 1 \).

Therefore, this separable basis was used in the extended preprint [GST16], but since it does not handle irregular grids well, we implemented and used the triangular shape functions described in [OP92, Section 7.3.1] for the published version [GST17] of the same paper (see Section 2.4 of Appendix K). This approach does the desired interpolation also for irregular grids.

**Remark 5.** Our software implementation described in Appendix J.6.5.1 can identify perturbed rectangular meshes like those in Figures I.2(c) and I.4 as long as no point in the fine mesh is perturbed more than half-way to the next row or column. However, since the interpolation property of the basis functions is only approximative for perturbed meshes, it is nonetheless recommended to run the code only for small perturbations of rectangular meshes.

**Remark 6.** As described in (I.5), we do updating of parameters \( X^0_p \) to new values \( X_p = X^0_p (1 - a_p) \). For the described regularization with tent functions, if the initial parameter values \( X^0_p \) on the fine grid are obtained from linear interpolation of the parameter values on the coarse grid, then the same linear interpolation property is in general **not** inherited by the parameter values \( X_p \). For instance, for the 1D structure in Figure I.2(a), if

\[
X^0_8 = X^0_7 + (X^0_{10} - X^0_7) \frac{x_8 - x_7}{x_{10} - x_7} \quad \text{and} \quad a_8 = a_7 + (a_{10} - a_7) \frac{x_8 - x_7}{x_{10} - x_7},
\]

Then the updated parameter value is

\[
X_8 = X^0_8(1 - a_8) = \left( X^0_7 + (X^0_{10} - X^0_7) \frac{x_8 - x_7}{x_{10} - x_7} \right) \cdot \left( 1 - a_7 - (a_{10} - a_7) \frac{x_8 - x_7}{x_{10} - x_7} \right).
\]

\[
= X^0_7(1 - a_7) - X^0_7(a_{10} - a_7) \frac{x_8 - x_7}{x_{10} - x_7} + \left( X^0_{10} - X^0_7 \right)(1 - a_7) \frac{x_8 - x_7}{x_{10} - x_7}.
\]

\[
= X_7 + \left( -X^0_7(a_{10} - a_7) + (X^0_{10} - X^0_7)(1 - a_7) \right) \frac{x_8 - x_7}{x_{10} - x_7}.
\]

Therefore, \( X_8 = X_7 + \left( X_{10} - X_7 \right)(a_{10} - a_7) \frac{x_8 - x_7}{x_{10} - x_7} \) only if \( X^0_{10} = X^0_7 \). Thus the physical parameters \( X_p \) inherit the linear interpolation property from the dimensionless parameters \( \alpha_p \) only if the initial values \( X^0_p = X^0 \) are constants not depending on \( p \).
I.4 Convex formulation of the optimization problem

The results for the Kirchhoff plate in Appendix G were based on a convex reformulation of the optimization problem. For the SBUF project, the goal has been to implement these methods and demonstrate how they work, which is done in Appendix G. In this section, we describe the theory behind this formulation briefly and, for simplicity, mainly for the frequency residual (I.6), but with a shorter note on how we have also generalized to mode shape residuals and multiple eigenvalues.

At iteration \( i + 1 \), the equation (I.18) for the free vibrations of an undamped structure is

\[
(K^{i+1} - \lambda^{i+1}_m M^{i+1})\phi^{i+1}_m = 0 \quad \text{with} \quad \phi^i_k T M^i \phi^i_m = \delta_{k,m}, \quad K, M \text{ symmetric.} \tag{I.31}
\]

Here \( \lambda^{i+1}_m \) denotes the \( i + 1 \)th iteration value of the eigenvalue \( \lambda^i_m \) in (I.6). In our MATLAB iterative implementation for convex optimization, we use same frequency residual \( r^i_m = r^i_m^a \) as in (I.6), except for a sign change. Moreover, the weights of the objective function \( f(a^i) \) are now included in the residual:

\[
f(a^i) = \frac{1}{2} r^i_m^T r^i_m = \frac{1}{2} \| r^i_m \|^2, \quad \text{with} \quad (r^i_m)_m = w_m \frac{\lambda^\text{mea}_m - \lambda^i_m}{\lambda^\text{mea}_m} = w_m \left( 1 - \frac{\lambda^i_m}{\lambda^\text{mea}_m} \right), \tag{I.32}
\]

for \( m = 1, \ldots, M \). We assume that the mass remains unchanged for the updated model and introduce the notation

\[
K^{i+1} = K^i + \Delta K^i, \\
M^{i+1} = M^i = M, \\
\lambda^{i+1}_m = \lambda^i_m + \Delta \lambda^i_m, \\
\phi^{i+1}_m = \phi^i_m + \Delta \phi^i_m. \tag{I.33}
\]

Note also that it follows from (I.31) that \( \phi^i_m^T K^i \phi^i_m = \phi^i_m^T \lambda^i_m M^i \phi^i_m = \lambda^i_m \). Substitution of this and (I.33) into (I.31) and multiplying from the left with \( \phi^{i+1}_m T \) gives

\[
0 = \phi^{i+1}_m^T (K^{i+1} - \lambda^{i+1}_m M) \phi^{i+1}_m = \phi^{i+1}_m^T (K^i + \Delta K^i) \phi^{i+1}_m - \lambda^{i+1}_m \\
= \phi^{i+1}_m^T K^i \phi^{i+1}_m + \phi^{i+1}_m^T \Delta K^i \phi^{i+1}_m - \lambda^{i+1}_m \\
= \phi^{i+1}_m^T K^i \phi^{i+1}_m + \phi^{i+1}_m^T \Delta K^i \phi^{i+1}_m + 2 \phi^{i+1}_m^T \Delta K^i \phi^{i+1}_m + \Delta \phi^{i+1}_m^T \Delta K^i \phi^{i+1}_m - \lambda^{i+1}_m, \\
\phi^i_m^T \Delta K^i \phi^i_m = \lambda^{i+1}_m - \phi^i_m^T K^i \phi^i_m - 2 \phi^i_m^T \Delta K^i \phi^i_m - \Delta \phi^i_m^T \Delta K^i \phi^i_m \\
= \lambda^{i+1}_m - \phi^i_m^T K^i \phi^i_m - 2 \phi^i_m^T \Delta K^i \phi^{i+1}_m - \Delta \phi^i_m^T \Delta K^i \phi^{i+1}_m \\
- 2 \phi^i_m^T \Delta K^i \phi^i_m - \Delta \phi^i_m^T \Delta K^i \phi^{i+1}_m \\
= \lambda^{i+1}_m - \lambda^i_m - 2 \phi^i_m^T \Delta K^i \phi^{i+1}_m - \Delta \phi^i_m^T \Delta K^i \phi^{i+1}_m - 2 \phi^i_m^T \Delta K^i \phi^{i+1}_m - \Delta \phi^i_m^T \Delta K^i \phi^{i+1}_m \\
= \lambda^{i+1}_m - \lambda^i_m - (\phi^i_m + \phi^{i+1}_m)^T K^i \phi^{i+1}_m - (\phi^i_m + \phi^{i+1}_m)^T \Delta K^i \phi^{i+1}_m. \tag{I.34}
\]

Based on experience that there usually is much smaller changes of the mode shapes than the changes of the frequencies, we will now assume for all \( m = 1, \ldots, M \) that

\[
\lambda^{i+1}_m - \lambda^i_m \gg (\phi^i_m + \phi^{i+1}_m)^T K^i \Delta \phi^{i+1}_m.
\]

From this and (I.32) we get the approximation

\[
\phi^i_m^T \Delta K^i \phi^i_m \approx \lambda^{i+1}_m - \lambda^i_m = -\lambda^\text{mea}_m \left( \frac{\lambda^i_m}{\lambda^\text{mea}_m} - 1 + 1 - \frac{\lambda^{i+1}_m}{\lambda^\text{mea}_m} \right) = -\lambda^\text{mea}_m \left( (r^i_m)_m - (r^{i+1}_m)_m \right). \tag{I.34}
\]
I.4 Convex formulation of the optimization problem

This corresponds to how a similar relationship was derived in [XH03], but as noted in [FG17], it can also be seen as a linearization using derivatives obtained from the Nelson or Fox Kapoor formulae. We have used the same linearization together with corresponding formulae for the derivatives computed with the Nelson method in our MATLAB interpretation, which therefore also works for mode shapes and multiple eigenvalues. It does however currently not work for the structures for which we use Abaqus, because then the matrices delivered by Abaqus gives problems with a singular matrix which is an important problem to fix in future research.

In the context of damage identification, the initial stiffness matrix $K^0$ is chosen to be the stiffness matrix obtained for the undamaged structure. As in previous sections, we consider a structure that is divided into $P$ groups of elements, with initial elasticity modulus $E^0_p$ for the $p$th group. The expanded order stiffness matrix for the $p$th group of elements is denoted $K^0_p$.

Thus we can write

$$K^0 = \sum_{p=1}^{P} K^0_p = \sum_{p=1}^{P} E^0_p \frac{K^0_p}{E^0_p}$$

(I.35)

for the $i$th iteration we denote the updated elasticity modulus and define the resulting stiffness matrix to be

$$K^i \triangleq \sum_{p=1}^{P} E^i_p \frac{K^0_p}{E^0_p} = \sum_{p=1}^{P} K^i_p$$

(I.36)

Insertion in (I.33) gives that

$$\Delta K^i = K^{i+1} - K^i = \sum_{p=1}^{P} K^{i+1}_p - K^i_p = \sum_{p=1}^{P} \frac{E^{i+1}_p - E^i_p}{E^0_p} K^0_p$$

$$=- \sum_{p=1}^{P} \left( \frac{E^0_p - E^{i+1}_p}{E^0_p} - \frac{E^0_p - E^i_p}{E^0_p} \right) K^0_p = - \sum_{p=1}^{P} (\Delta a^i)_p K^0_p,$$

(I.37)

where $\Delta a^i = a^{i+1} - a^i$ and $a^i_p \triangleq (a^i)_p = \frac{E^0_p - E^i_p}{E^0_p}$ is the $p$th element of the damage index (parameter) vector $a$, as in (I.5). Note also that Equations (I.35) and (I.36) give

$$K^i = K^0 - \sum_{p=1}^{P} a^i_p K^0_p.$$  

(I.38)

Now we define the $M \times P$-matrix $S^i = S(a^i)$ with elements

$$S^i_{m,p} = \frac{w_m}{\lambda^i_m} \phi^i_m \phi^i_m.$$  

(I.39)

Then insertion of (I.37) into (I.34) gives

$$(r^{i+1}_f)_m - (r^i_f)_m \approx - \frac{w_m}{\lambda^i_m} \phi^i_m \phi^i_m \Delta K^i \phi^i_m = \sum_{p=1}^{P} (\Delta a^i)_p \frac{w_m}{\lambda^i_m} \phi^i_m \phi^i_m K^0_p \phi^i_m = \sum_{p=1}^{P} S^i_{m,p} (\Delta a^i)_p$$

Hence,

$$S^i \Delta a^i = r^{i+1}_\lambda - r^i_\lambda,$$

(I.40)

Moreover, the Fox-Kapoor formula (I.17a) gives

$$\frac{\partial \lambda^i_m}{\partial a^i_p} = \phi^i_m \phi^i_m = - \phi^i_m K^0_p \phi^i_m.$$  

(I.41)
I implemented FEM updating methods

Insertion of this and (I.32) in (I.39) gives that

$$S_{m,p}^i = - \frac{w_m}{\lambda^i_{m \text{ma}}} \frac{\partial \lambda^i_m}{\partial a^i_p} = \frac{\partial (r^i_m)}{\partial a^i_p}. $$

The matrix $S$ for the partial derivatives of residuals with respect to the updating parameters is also known as the sensitivity matrix.

Using Equation (I.40) the residual can be linearized as follows

$$r^i_{\lambda} + 1 = S^i \Delta a^i + r^i_{\lambda}. $$

To find the updating parameters we minimize $1/2 \| r(a) \|_2^2$ by solving in each iteration the linearized minimization problem

$$\min_{\Delta a^i \in \mathbb{R}^n, l \leq \Delta a^i \leq u} \left\| S^i \Delta a^i + r^i_{\lambda} \right\|_2$$

for $\Delta a^i$. Then, in each iteration step the updating parameter vector is updated as $a^{i+1} = a^i + \Delta a^i$. This is the same minimization problem as in [WPP09, Eq. (19)]. Moreover, it is a convex problem in the sense that for any fixed $i$, the set

$$\Omega = \{ \Delta a^i | S^i \Delta a^i = -r^i_{\lambda} \}$$

is convex. In fact, for any $t \in [0,1]$ and $\Delta a^i_1, \Delta a^i_2 \in \Omega$ we have

$$S^i(t \Delta a^i_1 + (1-t) \Delta a^i_2) = t S^i \Delta a^i_1 + (1-t) S^i \Delta a^i_2 = -tr^i_{\lambda} - (1-t)r^i_{\lambda} = -r^i_{\lambda},$$

so that $t \Delta a^i_1 + (1-t) \Delta a^i_2 \in \Omega$ and hence, $\Omega$ is convex.

### I.4.1 Sparse $l^1$-norm regularization

Recall from after (I.37) that the damage index $a$ contains damage indices $a^i_p \overset{\text{def}}{=} \frac{E^0_p - E^i_p}{E^0_p}$ that ideally should be positive for damaged (groups of) elements and zero elsewhere. Moreover, damages in a structure are usually very well localized, so we are looking for minimization problem solutions with a small number of elements, that is, $a$ should ideally be a sparse vector.

To obtain sparseness also for noisy vibration measurements, one standard solution A standard solution of this problem is to use a regularized version of the minimization problem (I.41):

$$\min_{a \in \mathbb{R}^n, l \leq a \leq u} \left\| S^i \Delta a^i + r^i_{\lambda} \right\|_2 + \lambda R(\Delta a^i),$$

where $\lambda$ and $R$ are the regularization parameter and the regularization function, respectively. The regularization function should somehow measure deviations from desired properties of the solution, such as smoothness, sparsity, etc.

The most simple and intuitive measure of sparsity of vector $x$ is to count the number of nonzero elements, or using, so-called, $l_0$-"norm"

$$\| x \|_0 = \# \{ i : x_i \neq 0 \}. $$

It is not really a norm, since it does not satisfies the homogeneity property. The $l_0$-norm regularization problem belongs to the class of combinatorial problems, which are computationally
I.4 Convex formulation of the optimization problem

It is common practice to instead use its closest convex relaxation, the $l^1$-norm:

$$\min_{\Delta a^{(k)} \in \mathbb{R}^n \mid \Delta a^{(k)} \leq u} \left\| S^{(k)} \Delta a^{(k)} + r^{(k)}_\lambda \right\|_2^2 + \lambda \left\| \Delta a^{(k)} \right\|_1,$$

(I.43)

where $\left\| \Delta a^{(k)} \right\|_1$ is often called sparsifying term. Regularization with $l^1$-norm leads to sparse solution with only few nonzero elements [BV04]. There are different technical sufficient conditions under which the solution of the $l^1$-norm regularization coincide with the solution of the $l_0$-norm regularization, and thus (for the right choice of $\lambda$) is guaranteed to be optimally sparse. See, for instance, [TKS13], for a lengthy discussion and further references.

In next subsection we will describe some generalizations of this minimization problem that are used for the Kirchhoff plate in Appendix G. Some other alternatives to the $l^1$-norm regularization are investigated in [GST17], included as Appendix K.

I.4.2 Sparse regularization with $l^1$-norm and dictionaries

In the minimization problem (I.43), the second term is expected to favour solutions with most elements being (nearly) zero in the change $\Delta a^i$ in the updating parameter vector between two iterations. However, we would rather want the final solution $a^I$ to be sparse, so we will make one such modification below. Moreover, a sparse $a^I$ seems reasonable for point damages and small damages, for which $a$ already is sparse in the sense of having only one or a few nonzero elements.

Another possibility is that we have a so-called dictionary of vectors $\psi_n$ carefully chosen so that for typical damages, $a$ can be sparsified by decomposing it into a sum $a = \sum_n c_n \psi_n$ with most coefficients $c_n$ being (nearly) zero. Let the vectors $\psi_n$ be stored as column vectors in a dictionary matrix $\Psi$. We will in the following require $\Psi$ to have full row rank. The vectors $\psi_n$ are then called called a frame for their span (with certain associated so-called frame bounds closely related to the condition number of $\Psi$ [Gri02]), or a basis for their span if they are linearly independent (and thus $\Psi$ invertible). With notation $M^*$ for the complex conjugate transpose of a matrix $M$, we can define the analysis mapping with standard matrix $\Psi^*$ and the so-called canonical dual frame

$$\tilde{\Psi} = (\Psi \Psi^*)^{-1} \Psi,$$

(I.44)

which can be used to compute minimum $l^2$-norm coefficients for $a = \Psi c$:

$$c = \tilde{\Psi}^* a \quad \Rightarrow \quad a = \Psi c = \sum_n c_n \psi_n$$

(For the special case of a square matrix $\Psi$ with orthonormal column vectors, we get $\tilde{\Psi} = \Psi$.) For an overcomplete frame (that is, not linearly independent), there are infinitely many $b$ such that $a = \Psi b$, but (I.45) gives the minimum $l^2$-norm coefficients, that is $\sum_n |c_n|^2 \leq \sum_n |b_n|^2$, see for instance [Gri02, Theorem 1.46]. Thus, we can not use $\tilde{\Psi}$ for computing coefficients in the following and in the corresponding MATLAB implementation, because we will now rather want minimum $l^1$-norm coefficients. For more about the theory of frames and bases, see , for example, [Grö00, Chr02, Gri02].

Insertion of $\Delta a^i = \Psi c$ in the optimization problem (I.43) gives

$$\min_{\Delta a^i \in \mathbb{R}^n \mid \Delta a^i \leq u} \left\| S^{(k)} \Psi c + r^{(k)}_\lambda \right\|_2^2 + \lambda \left\| \Psi c \right\|_1.$$

We will only consider real-valued matrices on the following, but kept standard notation here, which also allows for complex-valued matrices and vectors.
Now if we instead want to keep the coefficients \( c = \Psi^* \Delta a^i \) sparse, we can change the regularization term as follows:

\[
\min_{c \in \mathbb{R}^n : \|c\|_1 \leq u} \| S^k \Psi c + r^i \|_2^2 + \lambda \|c\|_1. \tag{I.46}
\]

This does however not always make the final solution \( a^i \) sparse after \( I \) iterations, as shown in Figure G.11. Thus we modify the regularization term in (I.46) one step further to

\[
\min_{c \in \mathbb{R}^n : \|c\|_1 \leq u} \| S^k \Psi c + r^i \|_2^2 + \lambda \|b\|_1 \quad \text{with} \quad \Psi b = a^{i+1} = a^i + \Psi c. \tag{I.47}
\]

**Remark 7.** Some more work is needed to formally justify that (I.47) still is a convex formulation of the optimization problem. This is out of the scope of the SBUF project, but as we have understood it so far, a necessary and sufficient condition \( \|b\|_1 \) is a convex function of \( c \), where, for fixed \( a^i, b \) is a minimum \( l^1 \)-norm solution of

\[
\Psi b = a^{i+1} = a^i + \Psi c.
\]

Suppose for \( c_1 \) and \( c_2 \) that \( b_k \) is a minimum \( l^1 \)-norm solution of \( \Psi b_k = a^i + \Psi c_k \). For \( 0 \leq t \leq 1 \), let \( c = tc_1 + (1-t)c_2 \) and let \( b \) be a minimum \( l^1 \)-norm solution of

\[
\Psi b = a^i + \Psi c = a^i + \Psi (tc_1 + (1-t)c_2) = t(a^i + \Psi c_1) + (1-t)(a^i + \Psi c_2)
\]

\[
= t \Psi b_1 + (1-t) \Psi b_2 = \Psi (tb_1 + (1-t)b_2)
\]

Since \( b \) is the minimum \( l^1 \)-norm solution, we get that

\[
\|b\|_1 \leq \|tb_1 + (1-t)b_2\|_1 \leq t \|b_1\|_1 + (1-t) \|b_2\|_1.
\]

This proves the proposed convexity.

In this proof, we could not assume a unique \( b_k \) is a minimum \( l^1 \)-norm solution, this is an important difference from the unique \( l^2 \)-norm solution, as shown by the following two examples.

**Example 1** (Minimum \( l^p \)-norm solutions for \( p = 0, 1, 2 \)). Consider the following frame consisting of three vectors in \( \mathbb{R}^2 \) with angles \( \frac{2\pi}{3} \) between

\[
\Psi = \begin{pmatrix} 0 & \frac{\sqrt{3}}{2} \\ 1 & \frac{1}{2} \end{pmatrix}
\]

(sometimes called the Mercedes–Benz frame). A standard solution of \( \Psi c = (\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}) \), with Gaussian elimination gives that

\[
c = \begin{pmatrix} t \\ t \\ 1-t \end{pmatrix}
\]

and \( \|c_1\|_1 = |t| + |t| + |1-t| = \begin{cases} 1-3t & \text{if } t \leq 0, \\ 1+t & \text{if } 0 < t \leq 1, \\ 3t-1 & \text{if } t > 1. \end{cases} \]

The minimum \( l^1 \)-norm solution \( c_1 \) follows for \( t = 0 \), and coincides with the minimum \( l^0 \)-norm solution \( c = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \). in this case. The minimum \( l^2 \)-norm follows from (I.44):

\[
\tilde{\Psi} = (\Psi \Psi^*)^{-1} \Psi = \begin{pmatrix} 0 & \frac{-\sqrt{3}}{3} \\ \frac{2}{3} & \frac{1}{3} \end{pmatrix} \quad \text{and} \quad c = \tilde{\Psi}^* \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1/3 \\ 1/3 \end{pmatrix}.
\]
Example 2 (Minimum $l^p$-norm solutions for $p = 0, 1, 2$). Let

$$
\Psi = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.
$$

All solutions of $\Psi c = (\frac{1}{1})$ are $c = \begin{pmatrix} 1-t \\ t \\ 0 \end{pmatrix}$ for arbitrary $t \in \mathbb{R}$. Moreover, $\|c\|_1 \leq 1$, with equality for $0 \leq t \leq 1$, so there are infinitely many minimum $l^1$-norm solutions in this case, which coincide with two different minimum $l^0$-norm solutions for $t = 1$ and for $t = 0$ and with the unique minimum $l^2$-norm solution for $t = \frac{1}{2}$, since

$$
\tilde{\Psi} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}
$$

and the minimal $l^2$-norm solution is $c = \tilde{\Psi}^* \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

Choice of dictionaries

For the $20 \times 20$ element Kirchhoff plate in Appendix G, we assume that damages are expected to mainly consist of cracks directed along a direction parallel to the main diagonal from the lower left to the upper right corner. For such damages, we evaluate three different dictionaries of matrices, for which each matrix in the dictionary has all of its nonzero values on the main diagonal or on one of its side diagonals. It is more convenient here to consider dictionaries of $20 \times 20$-matrices, although the corresponding damage index matrices are reshaped into $400 \times 1$-vectors for the computations in the MATLAB implementation.

Kronecker Delta Dictionary corresponds to $\Psi$ being an identity matrix in (I.47), that is, the dictionary consists of 400 different matrices, each having one element equal to 1 and the others are 0.

Figure I.5: 100 of the 2870 matrices in the $45^\circ$ Cracks Dictionary. Blue dots for the nonzero elements. (Compared to Figure G.1 and G.3 the plates are upside down.)
This can be expected to be a good dictionary to detect point damages, since a point damage corresponds to only one nonzero coefficient.

**Haar Dictionary** Along each diagonal we choose a so-called Haar wavelet basis. It is slightly overcomplete (and then called a *frame* rather than a basis) for diagonals whose length is not an integer power of 2. Thus this dictionary is slightly overcomplete but close to the minimum possible dictionary size 400.

The way it is constructed, both big and small damages can be expected to possible to decompose into a sum of a small number of Haar basis functions. (see, for example, [Gri09, Section 1.1]). For example, at most \( \log_2(16) + 1 = 5 \) nonzero coefficients for any point damage.

**45° Cracks Dictionary** This dictionary consists of all “constant depth cracks” of length \( \geq 1 \) in the 45° direction, for example those on the main diagonal that are depicted in Figure 1.5 which is a highly overcomplete dictionary, consisting of 2870 matrices.
J The source code

This Appendix is meant as a “version 0” of a documentation of the software. Since the software still under development, parts of it is constantly changing, so the documentation here is mainly focused on some of the parts that were new for this project, whereas for some functions, we currently have the comments in the source code as documentation. The interested reader can however contact the authors for questions or a current updated versions of the software.

The source code in this appendix is provided under the following [Modified BSD License](#).

Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

J.1 Data storage

J.1.1 Shell, beam and spring elements

Figure J.1 (a) shows one group containing a square shell element and five strings, as well as one group containing a beam element and one triangular shell element. In the MATLAB files, we store these groups of elements as a cell array usually called `grPlemNodeNrs`, with one row for every group. For each shell and beam element, the node numbers are stored as a column vector. Spring elements are stored in two-column matrices, with node numbers in the first column and degree of freedom in the second column. The coordinates of the nodes are stored in a matrix.
usually called \texttt{nodeNrCoords} with each row containing the node number followed by the $x$-, $y$-
and $x$-coordinate of the node. For example, the groups of elements in Figure \ref{J.1} (a) can be 
stored as follows:

\[
grPlemNodeNrs = \begin{pmatrix}
1 \\
2 \\
4 \\
3 \\
10 \\
13
\end{pmatrix}, \quad \text{nodeNrCoords} = \begin{pmatrix}
1 & 0 & 1 & 0 \\
2 & 1 & 1 & 0 \\
3 & 0 & 0 & 0 \\
4 & 1 & 0 & 0 \\
10 & 3 & 1 & 0 \\
13 & 4 & 1 & 0 \\
9 & 3 & 0 & 0 \\
7 & 4 & 0 & 0
\end{pmatrix}.
\]

Here we could just as well have have stored all springs, for example, in a $5 \times 2$-matrix or in 
five $1 \times 2$-matrices. When groups contain different number of elements, the elements are stored 
from left to right in each row, and then shorter rows are padded with empty matrices on the 
right.

\section*{J.2 AbaqusPkg package}

The \texttt{AbaqusPkg} package contains methods for the interaction with Abaqus.

First short general description.

Then one subsubsection for each function in this package. Detailed description of the most 
important functions for our optimization problem. Short description in words of less crucial 
function.

\subsection*{J.2.1 The \texttt{AbaqusModalDataSim} class}

\textbf{Source code}

\begin{verbatim}
classdef AbaqusModalDataSim < SimulatorPkg.Simulator

properties (Access = private)
  updatingStructure % a kind of buffer for the updating data
  currDir
  datFilName
  inpFilName
  initInpFilName
  elemStiffMtxFilName
  nonUpdFileNames
  elemMtxJobStr

% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
end
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figJ1.png}
\caption{(a) The shell elements, beam elements and spring elements defined in \ref{J.1}.}
\end{figure}
function obj = AbaqusModalDataSim(initInpFilName,...
currDir,...
grNr2GrNameMap,...
nr0fDofsPerFemNode,...
P0,...
sparseMtx,...
DEBUG)

obj.initInpFilName = initInpFilName;
obj.currDir = currDir;

% clear old files

clear(obj);

obj.sparseMtx = sparseMtx;
obj.nr0fDofsPerFemNode = nr0fDofsPerFemNode;
obj.grNr2GrNameMap = grNr2GrNameMap;
%obj.AbaqusRun = false;

SlashIndex = find(initInpFilName=="/");
PointIndex = find(initInpFilName=="."); 
FileName = initInpFilName(SlashIndex(end)+1:PointIndex(end)-1);
obj.inpFilName = [FileName,'.inp'];
obj.datFilName = [FileName,'.dat'];
obj.elemStiffMtxFilName = upper([FileName,'.MTX']);

copyfile(initInpFilName,obj.inpFilName);

% NOTE! Here we run only to get mapping between local and global 
% node and element numbers! So no updating of input file is 
% needed!
run(obj,SimulatorPkg.ResultType.MODAL_DATA);

[obj.totNrOfNodes,...
obj.grNr2GrNameMap,...
obj.globElem2GrNrMap,...
obj.nonUpdFileNames,...
obj.elemMtxJobStr,...
obj.updatingStructure,...
obj.grElemNodeNrs,...
obj.globNodeCoords,...
obj.grCenterCoords] = ...
AbaqusPkg.AbaqusUtilities.ExtractModelParams(...
obj.datFilName,...
obj.inpFilName,...
obj.grNr2GrNameMap,...
DEBUG);

update(obj,P0,SimulatorPkg.ResultType.GROUP_STIFF_MTX);
run(obj,SimulatorPkg.ResultType.GROUP_STIFF_MTX);

function update(obj,P,type)
import AbaqusPkg.*

update(obj.updatingStructure_P,obj.grNr2GrNameMap);
if type==SimulatorPkg.ResultType.MODAL_DATA
%update(obj,updatingStructure_P,obj.grNr2GrNameMap);
AbaqusUtilities.updateInpFile(obj.inpFilName,...
obj.nonUpdFileNames,obj.updatingStructure);
```plaintext
% End of function clear.

function run(obj, type)

% Do nothing (no simulation needed, the info is in * . inp file)
end

function Result = getResult(obj, type)

% if just spring data is updating
end

function import AbaqusPkg =

% if
end

else if type==SimulatorPkg.ResultType.GROUP_STIFF_MTX

result = readStiffMtxFile(...
end

else if type==SimulatorPkg.ResultType.GLOBAL_NODE_COORDS

nodeNumbers = 1 : length(obj.globNodeCoords);
end

else if type==SimulatorPkg.ResultType.GROUP_ELEM_NODE_NRS

Result = obj.grElemNodeNrs;
end

else if type==SimulatorPkg.ResultType.GROUP_CENTER_COORDS

Result = obj.grCenterCoords;
end

end

function clear(obj)

SlashIndex = find(obj.initInpFilName=='/');
PointIndex = find(obj.initInpFilName=='.');
```
methods (Access = private)

function runAbaqus(obj)
  t = clock;
  istatus = dos(['abaqus job=' object.inpFilName 'interactive ']);
  display(['Abaqus job is finished in ' num2str(etime(clock,t)/60) ' minutes with status: ' num2str(istatus)]);
  if istatus==0
    %obj.AbaqusRun = true;
  end
end

end

%J.2.2 The AbaqusUtilities class

Source code

classdef AbaqusUtilities
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se

methods (Static)

  % function value = setRefFemNodeIndex(MasterFemGlobNodes,Mea2FemDofs,RefFemNode,RefFemDoF)
  % if RefFemDoF=0
  %   % Choose nPlateExcFrc as reference degree of freedom for normalizing
  %   % phiFEM and phiMeas in the function ModeShapeResid
  %   % ind = find(MasterFemGlobNodes==RefFemNode);
  %   % tmp = sum(Mea2FemDofs(1:ind-1,:)^2)==0);
  %   % nPlateExcFrc = tmp + find(Mea2FemDofs(tmp,:)==RefFemDoF); % index of the excitation
  %   % nPlateExcFrc = (find(MasterFemGlobNodes==RefFemNode)-1)*length(.
  %   % OptConst.MasterFemDofs)+
  %   % find(OptConst.MasterFemDofs==RefFemDoF); % index of the excitation
  %   value = nPlateExcFrc;
  % else
  %   % Will make ModeShapeResid call ChooseRefAcc for choosing nRef:
  %   value = 0;
  % end

% INPUT:
% DatFilName - Abaqus " .dat " output file with mapping between local and global element numbers
% InpFilName - Abaqus " .inp " input file
% GrNr2GrNamesMap - group number to group names mapping

% OUTPUT:
% nrOfModes - total number of modes in the Abaqus model
% nrOfNodes - total number of nodes in the Abaqus model
% newGrNr2GrNamesMap - updated GrNr2GrNamesMap, in the sense that each group in this map is associated with material properties that each group in this map is associated with material properties
% globElemNr2GrNrMap - table indexed by global element numbers and with entries saying to which group each particular element belongs
% nonUpdFileNames - names of the temporary files containing nonupdating parts of the Abaqus input file
% elemMtxJobStr - string representing the job step
% related to computation of the
% element stiffness matrices
% updatingStructure – structure that contains possible updating
% part of the input file, i.e. material,
% spring and section
% grNr2ElemNodeNrs – a cell array with rows indexed by group
% number containing global node numbers listed elementwise as
% column vectors.
% This cell array contains the group topology (or geometry)
% that can be used in order to show the updating parameters
% values for the groups. This cell array is also used in order
% to check if any group contains both spring and non spring
% elements, that can not be updated in one group! It is possible
% since each spring definition contains a particular dof associated
% with this spring.
% globNodeCoords – Nx3 matrix with rows indexed by the global
% node number containing x, y and z FEM coordinates
% grCenterCoords – |[NrOfGroups x 3]| matrix indexed by the
% group number and containing the x, y and z coordinates of the
% center of mass of a group.

function [nrOfNodes , nrOfNodes , ...]
    ... ... ... ... ... ... ...
    nelsE ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ... ...
    globCo ... ... ... ... ...
    ... ... ... ... ...
    grCenterCo ... = ExtractModelParams(datFilName , ...)
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ... ... ...
    ... ... ......
J.2 AbaqusPkg package

123

115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141

ElemPattern = ' ∗ ( Element ) ' ;
E l e m e n t M a t r i x P a t t e r n = ' ∗ELEMENT (MATRIX) OUTPUT ' ;
SpringPattern = ' ∗ ( S p r i n g ) , \ s+e l s e t =([ a−zA−Z0−9 −]+) ' ;
FreqPattern = ' ∗ ( Frequency ) , e i g e n s o l v e r= ' ;
M at er ia l Pa tt er n = ' ∗\ s ∗ ( M a t e r i a l ) , \ s ∗name =([ a−zA−Z0−9 −]+) ' ;
ConductivityPattern = ' ∗( Conductivity ) ' ;
Damp ingPatte rn = ' ∗ ( Damping ) ' ;
Dens ityPatte rn = ' ∗ ( D e n s i t y ) ' ;
Elas ticPatte rn = ' ∗ ( E l a s t i c ) ' ;
E xp a n s i o n P a t t er n = ' ∗ ( Expansion ) ' ;
% Number p a t t e r n s
%NumPattern = ' [ − + ] ? ( [ 0 − 9 ] ∗ \ . [ 0 − 9 ] ∗ | [ 0 − 9 ] + ) ' ;
%ExpNumPattern = ' ( [ + − ] ? \ d+\.\d ∗ | \ . \ d+|\d+) ( [ eE ][+ −]?\ d+) ? ' ;
pntIndex = f i n d ( inpFilName== ' . ' ) ;
N o n U p d F i l e N a m e C o m m o n = inpFilName ( 1 : pntIndex −1) ;
NameDelimiter = ' ∗ ' ;
Tokens = cell ( 1 ) ;
Tokens {1} = ' I n s t a n c e ' ;
Tokens {2} = ' Node ' ;
Tokens {3} = ' Element ' ;
Tokens {4} = ' E l s e t ' ;
Tokens {5} = ' S e c t i o n ' ;
Tokens {6} = ' End I n s t a n c e ' ;
Tokens {7} = ' S p r i n g ' ;
Tokens {8} = ' M a t e r i a l ' ;
Tokens {9} = ' Frequency ' ;
Tokens {10} = 'MATRIX ' ;

143
144
145
146
147
148
149
150

pattern = [ I ns ta nc e Pa tt er n ' | ' NodePattern
'| ' ...
E l e m e n t M a t r i x P a t t e r n ' | ' ElemPattern ' | ' . . .
ElsetPattern
' | ' Sect ionPatte rn
'| ' ...
SpringPattern
' | ' M at er ia l Pa tt er n ' | ' FreqPattern ' | '
E n d I n s t a n c e P a t t e r n ' | ' Densi tyPatte rn ' | ' . . .
Elas ticPatte rn ' | ' C o n d u c t i v i t y P a t t e r n ' | ' . . .
Damp ingPatte rn ' | ' E x p a n s i o n P a t t e rn ' | ' . . .
SectionCommentPattern ] ;

152
153
154
155
156
157
158
159

% H e l p e r mappings
g r N r 2 G l o b E l e m N r s M a p = cell ( nrOfGroups , 1 ) ;
g l o b E l e m N r 2 G l o b N o d e N r s M a p = cell ( nrOfElems , 1 ) ;
s e t N a m e 2 M t r l A n d G l o b E l e m N r s M a p = cell ( 1 , 3 ) ;
% { 1 , 1 } − g l o b a l s e t names = ' l o c a l s e t
% name'+ d e l i m i t e r +' i n s t a n c e name '
% { 1 , 2 } − m a t e r i a l name i f such i s found
% { 1 , 3 } − g l o b a l e l e m e n t numbers

161
162
163
164
165
166

i f DEBUG
msg = s p r i n t f ( '### READING %s ###' , inpFilName ) ;
lenOfMsg = s i z e ( msg , 2 ) ;
msgStars = repmat ( '# ' , 1 , lenOfMsg ) ;
f p r i n t f ( '%s \n%s \n%s \n ' , msgStars , msg , msgStars ) ;
end ;

168
169
170
171
172
173
174

setCount = 0 ; % number o f s e t s i n f e model
totNrOfElems = 0 ; % number o f e l e m e n t s i n f e model
updPartInd = 1 ; % t h e same a s i n s t a n c e i n d e x i n i n p f i l e
% l a s t d e f i n e d m a t e r i a l b e f o r e i t i s saved i n t o UpdatingStructure
material = [ ] ;
% l a s t found s e c t i o n comment b e f o r e i t i s used i n t h e s e c t i o n d e f
sect ionComme nt = [ ] ;

176
177
178
179
180
181
182
183
184
185
186
187

n on Up dF i le Na me s { updPartInd , 1 } = s p r i n t f ( '%s PART %d ' , . . .
NonUpdFileNameCommon , updPartInd ) ;
fidNonUpd = f o p e n ( n on Up dF i le Na me s { updPartInd , 1 } , 'w ' ) ;
fid = f o p e n ( inpFilName ) ;
tline = f g e t l ( fid ) ;
w h i l e ischar ( tline ) % i f t l i n e i s a r r a y o f c h a r a c t e r s
%d i s p ( t l i n e ) % TODO −> l o g g e r ?
[ ˜ , tok ] = regexp ( tline , pattern , ' match ' , ' t o k e n s ' , ' i g n o r e c a s e ' ) ;
i f ˜ i s e m p t y ( tok )
index = f i n d ( strcmpi ( tok { 1 } { 1 } , Tokens ) ) ; % c a s e i n s e n s i t i v e !
i f ˜ i s e m p t y ( index )
switch index

...


case 1 % Reading instance name
nrOfElemsInInst = 0;
[tline, instName, instTranslationVector, ...
  instRotAngle, instRotAxisPnt1, ...
  instRotAxisPnt2] = ...;
readInstanceDef(tline, ...
fid, fidNonUpd, tok{1});
case 2 % Reading node numbers with coordinates
% * Node print is also processed here
[tline, globNodeInd, globCoords] = ...;
readNodeCoords(tline, ...
fid, fidNonUpd, ...
instName, ...
instLocal2GlobalNodeNrsMap, ...
instTranslationVector, ...
instRotAngle, ...
instRotAxisPnt1, instRotAxisPnt2);
if ~isempty(globNodeInd) && ~isempty(globCoords)
globNodeCoords(globNodeInd,:) = globCoords;
end
case 3 % Reading elements with node numbers
[tline, nrOfElems, globElemInd, globNodeNrs] = ...;
readElemNodeNrs(tline, ...
fid, fidNonUpd, ...
instName, ...
instLocal2GlobalElemNrsMap, ...
instLocal2GlobalNodeNrsMap);
for i = 1:length(globElemInd)
globElemNr2GlobNodeNrsMap{globElemInd(i)} = ...;
globNodeNrs{1,i} = globNodeNrs{1,i};
case 4 % Reading elements in a elset (both descriptive/generate)
[tline, instNameTmp, localSetName, globElems] = ...;
readElsetElemNrs(...
tline, fid, fidNonUpd, tok{1}, instName, ...
instLocal2GlobalElemNrsMap);
fINUEI{localsetName ... NameDelimiter ...
instNameTmp];
setCount = setCount + 1;
localSetName2MtrlAndGlobElemNrsMap{setCount,1} = ...
fullLocalSetNamel = [localSetName ...
NameDelimiter ...
locallsetName];
setName2MtrlAndGlobElemNrsMap{setCount,3} = ...
globElems;
grIndex,] = find(strcmp(grNr2GrNamesMap, ...
localSetName));
if ~isempty(grIndex)
  if numel(grIndex)==1
    grNr2GlobElemNrsMap{grIndex} = union(...
grNr2GlobElemNrsMap{grIndex}, globElems);
  else
    error(’Groups {index= %s} intersect by set {name= %s}’,
       grIndex ',', %d , ', localSetName);
end
case 5 % Reading section definition to connect
% set with material properties. Note! % Section comment if there is such is also
% read by ReadSectionDef function!
[sectionComment, localSetName, materialName, ...
updatlngStructure] = ...;
readSectionDef(tline, fid, ...
fidNonUpd, updatePartInd, tok{1}, ...
instName, updateIngStructure, ...
sectionComment);
if ~isempty(localSetName)
  updLocalSetNamel = [localSetName ...
NameDelimiter instName];
  % general part for Beam General and Beam
  % Section (what about shell?)
  [setIndex, ] = GeneralUtilities.FindCells(...}
updLocalSetName = ....
setName2MtrlAndGlobElemNrsMap(:,1));
setName2MtrlAndGlobElemNrsMap(setIndex,2) = ...
materialName;
end
time = fgetl(fid);
case 6 % Reading end instance
  instName = [];
  instTranslationVector = [];
  instRotAngle = [];
  totNrOfElems = totNrOfElems + nrOfElemsInInst;
  updPartInd = updPartInd + 1;
  fclose(fidNonUpd);
  nonUpdFileNames{ updPartInd,1 } = sprintf(...
    '%s_PART%d', NonUpdFileNamesCommon, ...;
  updPartInd);
  fidNonUpd = fopen(nonUpdFileNames{ updPartInd, 1 }, 'w');
  fprintf(fidNonUpd, '%d', time);
  fclose(fidNonUpd);
end

case 7 % Reading Spring definition
[tLine, updatingStructure, spring, globNodes] =...;
globElems, SpringDof] =...;
readSpringDef(tLine, fid,...;
updPartInd, instName,...;
instLocal2GlobalElementNrsMap,...;
instLocal2GlobalNodeNrsMap,...;
updateStructure);
for i = 1:length(globElems)
globElemsNr2GlobalNodeNrsMap = globElems(i) = ...
union(globElemsNr2GlobalNodeNrsMap{...}
gelemElems(i), globNodes(i));
end

elsetName = tok{1}{2};
[grIndex, ""] = GeneralUtilities.FindCells(...
  elsetName, grNr2GrNamesMap);
if isempty(grIndex)
  springName = elsetName; % name is unique by definition of ←
  the group name
  if numel(grIndex) == 1
    N = size(grNr2ElemNodeNrs{grIndex},2);
    n = numel(globNodes);
    nodeDof = [globNodes SpringDof ones(n,1)];
    %nodeDof = [globNodes
    % SpringDof ones(1,n)];
    grNr2ElemNodeNrs{grIndex,N+1} = nodeDof;
    grNr2GlobalElementNrsMap{grIndex} = union(...
    grNr2GlobalElementNrsMap(grIndex), globElems);
  else
    error('Groups {index=%s} intersect by spring {name=%s←
    }',...,
    num2str(grIndex, ', ', '), springName);
  end
else
  springName = [elsetName NameDelimiter instName];
end
setName(spring, springName);
setCount = setCount + 1;
setName2MtrlAndGlobElemNrsMap(setCount,1) = ...
springName;
setName2MtrlAndGlobElemNrsMap(setCount,3) = ...
globElems;
case 8 % Reading material name
[tLine, fidNonUpd, updPartInd, nonUpdFileNames, material, ...
  materialName] =...;
readMaterialDef(fid, fidNonUpd,...
tok{1}, updPartInd,...
updateStructure,...
nonUpdFileNames,...
NonUpdFileNamesCommon,...
material);
[grIndex, ""] = GeneralUtilities.FindCells(...
  materialName, grNr2GrNamesMap);
if isempty(grIndex)
if numel(grIndex)==1
    [setIndex,""] = GeneralUtilities.FindCells(...
        materialName,...
        setName2MtrlAndGlobElemNrsMap(:,2));
    GlobElemNr = cell2mat(...
        setName2MtrlAndGlobElemNrsMap(setIndex,3));
    GlobElemNr = unique(GlobElemNr);
    grNr2GlobElemNrsMap{grIndex} = union(...
        grNr2GlobElemNrsMap{grIndex},GlobElemNr);
else
    error('Groups {index=%s} intersect by material {name←%s}...
        num2str(grIndex,'%d'),materialName);
e end
end
%tline = fgets(fid);
case 9 % Reading total number of mode shapes in FEM model
    fprintf(fidNonUpd,'%s
        nrOfModes,tline]=...  
        readTotNrOfModes(tline,...  
        fid,fidNonUpd);
case 10 % Reading element matrix output definition
    elemMtxJobStr = tline;
    tline = fgets(fid);
end; % switch
else
    fprintf(fidNonUpd,'%s
        tline = fgets(fid);
    end; % switch
if DEBUG
    msg = sprintf('# # # READING %s FILE IS FINISHED # # #',inpFilName);
    lenOfMsg = size(msg,2);
    msgStars = repmat('#',1,lenOfMsg);
    fprintf('%s
        msgStars,msg,msgStars);
e end
% Define GrNr2ElemNodeNrs and GroupCenterCoords
% Up to here only spring groups are filled in GrNr2ElemNodeNrs
% Other groups are defined in 2 cell arrays:
% 1. GrNr2GlobElemNrsMap
% 2. GlobElemNr2GlobNodeNrsMap
for grIndex=1:.nrOfGroups
    S = zeros(1,3);
    nrOfNodesInGroup = 0;
    grGlobElemNrs = grNr2GlobElemNrsMap{grIndex};
    N = numel(grGlobElemNrs);
    if isempty(grNr2ElemNodeNrs{grIndex})
        for i=1:N
            nodeNrs = globElemNr2GlobNodeNrsMap{globElemNrs(i)};
            grNr2ElemNodeNrs{grIndex,i} = nodeNrs(:,);
            nrOfNodesInGroup = nrOfNodesInGroup + size(nodeNrs,2);
            S = S + sum(globNodeCoords(nodeNrs));
        end
        else
            % group of springs case
            maxNrOfElems = size(grNr2ElemNodeNrs,2);
            grGlobNodeNrs = [];
            for ElemNr=1:maxNrOfElems
                nodeNrs = grNr2ElemNodeNrs{grIndex,ElemNr};
grGlobNodeNrs = [grGlobNodeNrs nodeNrs];
if isempty(nodeNrs)
    nrOfNodesInGroup = nrOfNodesInGroup + size(nodeNrs,2);
    S = S + sum(globNodeCoords(nodeNrs));
end
for i=1:N
    nodeNrs = globElemNr2GlobNodeNrsMap{grGlobElemNrs(i)};
    if all(ismember(nodeNrs,grGlobNodeNrs(1,:)))
        error('Erroneous group {index=%d} definition: %s %s'
               'attempt to update spring %s together with material elasticity modulus!');
    end
end
gcrCenterCoords(grIndex,:) = S/nrOfNodesInGroup;
end

%%%% Check if the updating groups are defined neither by material
%%%% nor by spring name. If this is a case redefine these groups
%%%% in terms of material names.
%%%% Output parameter
newGrNr2GrNamesMap = UpdateGrNr2GrNameMap(...
    setName2MtrlAndGlobElemNrsMap, ...
    grNr2GrNamesMap, ...
    NameDelimiter, ...
    updatingStructure, ...
    grNr2GlobElemNrsMap, ...
    DEBUG);

%%%% Create a simple map from global element number to group number
%%%% Output parameter
globElemNr2GrNrMap = zeros(totNrOfElems,1);
%%%% only for predefined groups??
for grIndex=1:nrOfGroups
    ElemNr = grNr2GlobElemNrsMap{grIndex};
    globElemNr2GrNrMap(ElemNr) = grIndex*ones(size(ElemNr,2),1);
end

%%%% Update Abaqus input file by reading non updating files and
%%%% corresponding updating parts stored in UpdatingStructure.
%%%% Note! For each instance (non updating files and UpdatingStructure
%%%% were built based on the instance partition of FE model) first the
%%%% material definitions are written in the input file, then non
%%%% updating part corresponding to this instance and finally section
%%%% definitions followed by spring definitions are written.
function updateInpFile(InpFilName,NonUpdFileNames,UpdatingStructure)
    NrOfNonUpdParts = size(NonUpdFileNames,1);
    fid = fopen(InpFilName,'w');    % open for reading and writing

    for i=1:NrOfNonUpdParts
        MaterialStr = mtrlList2String(UpdatingStructure,i);
        if ~isempty(MaterialStr)
            fprintf(fid,'%s',MaterialStr);
        end
        if size(NonUpdFileNames,1)>>i &~ isempty(NonUpdFileNames{i})
            fid = fopen(NonUpdFileNames{i},'r');
            tline = fgets(fid);
            while ischar(tline)
                fprintf(fid,'%s',tline);
                tline = fgets(fid);
            end;
            fclose(fid);
        end
    end
    UpdPartStr = [];
    SecStr = sectionList2String(UpdatingStructure,i);
    SpringStr = springList2String(UpdatingStructure,i);
```matlab
if isempty(SecStr)
    UpdPartStr = sprintf('%s',SecStr);
end
if isempty(SpringStr)
    UpdPartStr = sprintf('%s',UpdPartStr, SpringStr);
end
if isempty(UpdPartStr)
    fprintf(fid, '%s', UpdPartStr);
end
end

% CHANGE to system(‘type f1 f2 > f3’) 
fclose(fid);

function Ke = ReadElemStiffMtxFile(nofNodes, nofDofs, lookup, file, ...
SparseMtx)
  % Note we cannot compose the global stiffness matrix by using . mtx file ,
  % since it could happen that the element set associated with this file is
  % not complete , i.e. not cover the whole structure. Check for that the
  % Abaqus . inp file for the definition of ELEMENT MATRIX OUTPUT.
  % If we need the global stiffness and mass matrices it is a good idea to
  % retrieve them in separate files by saying this in the . inp file .
  
  % Example:
  % Ke = ReadElementStiffMtxFile(1500,6,[ones(365,1);2],’Matrix.txt’,1);
  % file = ’Matrix.txt’;
  % SparseMtx = 1;
  % nofNodes = 1500; % number of global nodes
  % nofDofs = 6; % Assume each node in the FE model has 6 dofs
  % nofGroups = 2;
  % lookup = ones(366,1);
  % lookup(366) = 2;

  import UtilityPkg.*
  import AbaqusPkg.*
  
  Msg = [’, file, ’ make take a while!’ ];
  LenOfMsg = size(Msg,2);
  MsgStars = repmat(’#’, 1, LenOfMsg);
  fprintf(’
%sn%s
%sn’, MsgStars, Msg, MsgStars);

  NonZeros = find(lookup);
  ElemCheckArr = zeros(size(NonZeros,1),2);
  ElemCheckArr(:,1) = NonZeros;

  ElementNumberPattern = ’**\s*(ELEMENT NUMBER)\s+(\d+)’;
  UserElemNodesPattern = ’USER ELEMENT, NODES=\s+(\d+)' ;
  ElementNodesPattern = ’**\s*(ELEMENT NODES) ’;
  StiffMatrixPattern = ’**\s*(MATRIX,TYPE=STIFFNESS) ’;
  IntNumberPattern = ’\d+’;
  MatrixElementsPattern = ’[\+-]?\d+\|\d+\|\d+\|\d+\|\d+\|\d+\|\d+\|\d+\|\d+\|\d+\|\d+\]([eE][\+-]?\d+)?’;

  Tokens = cell(1);
  Tokens{1} = ’ELEMENT NUMBER’;
  Tokens{2} = ’USER ELEMENT, NODES’;
  Tokens{3} = ’ELEMENT NODES’;
  Tokens{4} = ’MATRIX,TYPE=STIFFNESS’;

  nofGroups = max(lookup); % no. of groups

  KNofRows = nofNodes*nofDofs; % Nr of rows in the stiffness matrix
  if SparseMtx
    Ke = cell(nofGroups,1);
    for gg=1:nofGroups
      Ke{gg} = sparse(KNofRows, KNofRows);
    end
  else
    Ke = zeros(KNofRows, KNofRows, nofGroups);
  end
```

fid = fopen(file);
tline = fgets(fid);  % Read next line from file

pattern = ElementNumberPattern;
while ischar(tline)  % if tline is array of characters
    str tok = regexp(tline, pattern, 'match', 'tokens');
    if ~isempty(str)
        index = strmatch(tok{1}{1}, Tokens, 'exact');
    switch index
        case 1  % Reading element number line
            ElemNumber = str2num(tok{1}{2});
            GrNr = lookup(ElemNumber);
            pattern = UserElemNodesPattern;
        case 2  % Reading element nodes number line
            n operator = str2num(tok{1}{2});
            pattern = ElementNodesPattern;
        case 3  % Reading nodes in the current element
            NodesInElem = { };
            while ( length(NodesInElem) < n operator )
                tline = fgets(fid);
                NodesInElem{str trim(str)} = str2num( tline{1}{1} );
            end
            dofNrs = GeneralUtilities.cell2num(NodesInElem);
            LocalNofDofs = length(dofNrs);
            pattern = StiffMatrixPattern;
        case 4  % Reading element stiffness matrix
            KeNofRows = n operator InElem + LocalNofDofs;
            pattern = StiffMatrixPattern;
            while ( length(NodesInElem{1} < KeNofRows )
                tline = fgets(fid);
                Str = regexp( tline{1} );
                pattern = ElementNumberPattern;
            end
            KeLoc = zeros(KeNofRows);  % Element stiffness matrix
            % === Read the elements in and below the diagonal of KeLoc
            for RowNr = 1:KeNofRows
                tline = fgets(fid);
                MatrixElements = { };
                while ( length(MatrixElements{1} ) < RowNr )
                    tline = fgets(fid);
                end
                KeLoc(RowNr, 1:RowNr) = MatrixElements{1};
            end
            KeLoc = GeneralUtilities.Triang2SymmMtx(KeLoc);
            Dofs = LocalInd2GlobalInd(...
                NodesInElem, dofNrs, n operator Dofs);
            % Assemble the current element stiffness matrix at the right
            % place in the current element group stiffness matrix
            if GrNr = 0
                ind = find(ElemCheckArr(:, 1) = = ElemNumber);
                ElemCheckArr(ind, 2) = 1;
                if SparseMtx
                    Ke(GrNr)(Dofs, Dofs) = Ke(GrNr)(Dofs, Dofs) + KeLoc;
                else
                    Ke(Dofs, Dofs, GrNr) = Ke(Dofs, Dofs, GrNr) + KeLoc;
                end
            end
            pattern = ElementNumberPattern;
        end
    end
    tline = fgets(fid);  % Read next line from file
end
fclose(fid);

ind = find(ElemCheckArr(:, 2) == 0);
if isempty(ind)
    Msg = ['Reading file= ', file, ' has finished successfully!'];
else
    Msg1 = 'Fail. No element stiffness matrices have found!';
    fprintf('%s\n%s\n%s\n'.num2str(ElemCheckArr(ind), '%d', 'r'));
    Msg = ['Reading file=', file, ', has failed! '];
    error(Msg);
end
LenOfMsg = size(Msg, 2);
MsgStars = repmat('#', 1, LenOfMsg);
fprintf('%s\n%s\n%s\n'.num2str(MsgStars, Msg, MsgStars);
end

function ModalDataFEM = ReadFreqDatFile(file,...
    TotNrOfNodes ,..., 
    TotNrOfModes ,..., 
    NrOfDofsPerNode)
import ModalDataPkg.ModalData

% INPUT:
% file = name for the Abaqus " .dat ", file with eigenfrequencies and mode shapes
% TotNrOfNodes = number of nodes in the entire FE model
% TotNrOfModes = number of modes found for the FE model
% NrOfDofsPerNode = number of dofs for each node in the FE model

% OUTPUT:
% nuFEMcomp = TotNrOfModes number of eigenfrequencies for the entire FE model
% phiFEMcomp = TotNrOfModes number of mode shapes corresponding to nuFEMcomp and
% defined for the predefined set in the Abaqus " .dat " file

phiFEMcomp = zeros(TotNrOfNodes + NrOfDofsPerNode, TotNrOfModes);

EigenvalueOutputPattern = '%*s(E I G E N V A L U E O U T P U T)\n*%s';
EigenvalueNumberPattern = '%*s(E I G E N V A L U E N U M B E R)\n*%s';

Tokens = cell(1);
Tokens{1} = 'E I G E N V A L U E O U T P U T';
Tokens{2} = 'E I G E N V A L U E N U M B E R';

fid = fopen(file);
tline = fgets(fid);

%ModeNumber = 0; % mode shape number counter

pattern = EigenvalueOutputPattern;

while ischar(tline) % if tline is array of characters
    tline = strtrim(tline);
    [str tok] = regexp(tline, pattern, 'match', 'tokens');
    if isempty(str)
        %disp(str);
        index = find(strncmp(Tokens, tok{1}{1}));
    switch index
        case 1 % Reading eigenvalue output
            % skip 5 lines
            fgets(fid); fgets(fid); fgets(fid); fgets(fid); fgets(fid);
            str = fgets(fid);
            while isempty(str)
                data = textscan(str, '%s%f4%f4%f4%f4%4d4' );
                % read formatted data
                nuFEMcomp(sscanf(cell2mat(data{1}), '%d')) = ...
                data{4}; % frequency in cycles/time
                str = fgets(fid);
            end;
            pattern = EigenvalueNumberPattern;
        case 2 % Reading eigenvalue number
            ModeNumber = str2num(tok{1}{2});
            % skip 14 lines
            fgets(fid); fgets(fid); fgets(fid); fgets(fid); fgets(fid);
            fgets(fid); fgets(fid); fgets(fid); fgets(fid);
            fgets(fid); fgets(fid); fgets(fid); fgets(fid);
            fgets(fid); fgets(fid); fgets(fid); fgets(fid);
        ...
```matlab
fgetl(fid); fgetl(fid);
str = strtrim(fgetl(fid));
while ~isempty(str)
    data = textscan(str, '%d%e%e%f%f%f%f%f%f%f%f');
    phiFEMcomp((ind-1)*NrOfDofsPerNode+1:...
        [data{2} data{3} data{4} data{5} data{6} data{7}]);
    str = strtrim(fgetl(fid));
end;

ModeNumber = ModeNumber + 1;
case 0
    break;
end;
tline = fgetl(fid);
fclose(fid);
ModalDataFEM = ModalData(nuFEMcomp, phiFEMcomp, ...
    zeros(length(nuFEMcomp),1), (1:length(nuFEMcomp)));
end

tline = fgetl(fid);
data = regexp(tline, InstNameNumNumPattern, '
    names');
while ~strcmp(tline, '')
    if ~isempty(data)
        if strcmp(CurrentInstName, data.name) % instance change
            if isempty(CurrentInstName)
                GlobalNumsInInst = [GlobalNumsInInst ...
                    str2num(data.num2)];
            else
                MapInd = MapInd + 1;
                Map{MapInd,1} = CurrentInstName;
                Map{MapInd,2} = GlobalNumsInInst;
                CurrentInstName = data.name;
                GlobalNumsInInst = str2num(data.num2);
            end;
            else % filling the same instance with global element map
                GlobalNumsInInst = [GlobalNumsInInst ...
                    str2num(data.num2)];
            end;
        end;
    end;
    tline = fgetl(fid);
data = regexp(tline, InstNameNumNumPattern, '
    names');
end;

if MapInd=0
    % For the last found instance
    MapInd = MapInd + 1;
    Map{MapInd,1} = CurrentInstName;
    Map{MapInd,2} = GlobalNumsInInst;
    NoOfGlobals = max(Map{MapInd,2});
end
end

function [Map, NoOfGlobals] = ReadLocal2GlobalMap(fid, InstNameNumNumPattern)
CurrentInstName = [];
GlobalNumsInInst = [];
MapInd = 0;
tline = fgetl(fid);
data = regexp(tline, InstNameNumNumPattern, 'names');
while ~strcmp(tline, '')
    if ~isempty(data)
        if strcmp(CurrentInstName, data.name) % instance change
            if isempty(CurrentInstName)
                GlobalNumsInInst = [GlobalNumsInInst ...
                    str2num(data.num2)];
            else
                MapInd = MapInd + 1;
                Map{MapInd,1} = CurrentInstName;
                Map{MapInd,2} = GlobalNumsInInst;
                CurrentInstName = data.name;
                GlobalNumsInInst = str2num(data.num2);
            end;
            else % filling the same instance with global element map
                GlobalNumsInInst = [GlobalNumsInInst ...
                    str2num(data.num2)];
            end;
        end;
    end;
    tline = fgetl(fid);
data = regexp(tline, InstNameNumNumPattern, 'names');
end;

function newGrNr2GrNameMap = UpdateGrNr2GrNameMap(...
        setName2MtrlAndGlobalElemNrsMap, ...
    grNr2GrNameMap, NameDelimiter, ...
    updatingStructure, ...)
    grNr2GlobalElemNrsMap, ...
import UtilityPkg.*
import AbaqusPkg.*

[ nrOfGroups, nrOfSubgroups ] = size ( grNr2GrNameMap ) ;

nrOfSets = length ( setName2MtrlAndGlobalElemNrsMap ( :, 1 ) ) ;

ewGrNr2GlobElemNrsMap = cell ( nrOfGroups, 1 ) ;
newGrNr2GrNameMap = cell ( nrOfGroups, 1 ) ;

grNr2SetNameMap = cell ( nrOfGroups, 1 ) ;
fail = false ;

for i = 1 : nrOfGroups
    % for each group
    nrOfNewSubgroups = 0 ;
    nrOfNewSubsets = 0 ;
    newGroupGlobElemNrs = [ ] ;
    currentGrGlobElemNrs = [ ] ;
    for j = 1 : nrOfSubgroups
        % for each subgroup in a group associated with some name
        subgroupName = grNr2GrNameMap { i , j } ;
        if ~ isEmpty ( subgroupName )
            if isMaterialName ( updatingStructure , subgroupName )
                [ r , c ] = GeneralUtilities . FindCells ( subgroupName , ...,
                    newGrNr2GrNameMap ) ;
                if ~ isEmpty ( r ) || isnan ( r )
                    nrOfNewSubgroups = nrOfNewSubgroups + 1 ;
                    newGrNr2GlobElemNrs { i , nrOfNewSubgroups } = ...
                        grpGlobElemNrs ;
                    else
                        D = setdiff ( r , i ) ;
                        if ~ isEmpty ( D )
                            error ( [ 'Group { index= %d } intersects with groups ' ...
                                '{index= %s } by material { name= %s }' ] , ...
                                i , num2str ( D , ' %d , ' ) , subgroupName ) ;
                        end
                    end
                end
            elseif isSpringName ( updatingStructure , subgroupName )
                [ r , c ] = GeneralUtilities . FindCells ( subgroupName , ...,
                    newGrNr2GrNameMap ) ;
                if ~ isEmpty ( r ) || isnan ( r )
                    nrOfNewSubgroups = nrOfNewSubgroups + 1 ;
                    newGrNr2GlobElemNrs { i , nrOfNewSubgroups } = ...
                        subgroupName ;
                else
                    D = setdiff ( r , i ) ;
                    if ~ isEmpty ( D )
                        error ( [ 'Group { index= %d } intersects with groups ' ...
                            '{index= %s } by spring { name= %s }' ] , ...
                            i , num2str ( D , ' %d , ' ) , subgroupName ) ;
                    end
                end
            end
        end
    end
end

cellArray = setName2MtrlAndGlobalElemNrsMap ( : , [ 2 3 ] ) ;
% set name - global element numbers
[subgroupGlobElems , subgroupInd ] = ...
    FindSubgroupGlobElems ( ...,
        setName2MtrlAndGlobalElemNrsMap ) ;
newGroupGlobElemNrs = union ( newGroupGlobElemNrs , ...,
    subgroupGlobElems ) ;
currentGrGlobElemNrs = union ( ...,
    currentGrGlobElemNrs , subgroupGlobElems ) ;
newGlobElemNrs = newGlobElemNrs + 1 ;
grNr2SetNameMap { i , nrOfNewSubsets } = ...
    subgroupName ;
end
newGroupName, NameDelimiter, cellArray);  
newGroupGlobElemNrs = union(newGroupGlobElemNrs, ...  
subgroupGlobElems);  
currentGrGlobElemNrs = union(...  
currentGrGlobElemNrs, subgroupGlobElems);  
nrOfNewSubsets = nrOfNewSubsets + 1;  
grNr2SetNameMap{i, nrOfNewSubsets} = ...  
subgroupName; % for error handling only!  
else  
 if isSectionName(updatingStructure, subgroupName)  
 % HAS NOT BEEN TESTED YET!!!!  
 % retrieve properly global element numbers  
 % old and new  
 [r, c] = GeneralUtilities.FindCells(...  
 subgroupName, newGrNr2GrNameMap);  
 if isempty(r) || isnan(r)  
nrOfNewSubgroups = nrOfNewSubgroups + 1;  
newGrNr2GrNameMap{i, nrOfNewSubgroups} = ...  
subgroupName;  
else  
 D = setdiff(r, i);  
 if ~isempty(D)  
 error(['Group {index=%d} intersects with groups ' ...  
{index=%s} by general section {name=%s}'] ...  
i, num2str(D, ', ', '), subgroupName);  
end  
end  
nrOfNewSubsets = nrOfNewSubsets + 1;  
grNr2SetNameMap{i, nrOfNewSubsets} = ...  
subgroupName; % for the test only!  
else  
 cellArray = setName2MtrlAndGlobalElemNrsMap(:, [1 3]);  
% set name = set global element numbers  
[subgroupGlobElems, subgroupInd] = ...  
FindSubgroupGlobElemNbs (...  
subgroupName, NameDelimiter, cellArray);  
if ~isempty(subgroupInd)  
currentGrGlobElemNrs = union(...  
currentGrGlobElemNrs, subgroupGlobElems);  
% find a set with the same element numbers  
% as in the subgroup for which material  
% properties are defined.  
for k=1:nrOfSets % all sets  
arr1 = setName2MtrlAndGlobalElemNrsMap{k,3};  
str2 = regexp(setName2MtrlAndGlobalElemNrsMap{k,1} ...  
NameDelimiter, 'split');  
str2 = strtrim(str2);  
materialName = setName2MtrlAndGlobalElemNrsMap{k,2};  
if ~isempty(materialName)  
% Set has a material properties  
setName = setName2MtrlAndGlobalElemNrsMap{k,1}; % For the test ←  
only!  
if all(ismember(arr1, subgroupGlobElems))  
% Set is a subset of  
% current subgroup  
elseif strcmp(SubgroupName, str2{1}{1})  
[r, c] = GeneralUtilities.FindCells(...  
materialName, newGrNr2GrNameMap);  
if isempty(r) || isnan(r)  
nrOfNewSubgroups = nrOfNewSubgroups + 1;  
newGrNr2GrNameMap{i, nrOfNewSubgroups} = ...  
materialName;  
else  
 D = setdiff(r, i);  
 if ~isempty(D)  
 error(['Group {index=%d} intersects with groups ' ...  
{index=%s} by ' ...  
'set {name=%s}'] ...  
i, num2str(D, ', ', '), materialName);  
end  
end  
newGroupGlobElemNrs = union(...  
newGroupGlobElemNrs, arr1);
nrOfNewSubsets = nrOfNewSubsets + 1;
grNr2SetNameMap{i,nrOfNewSubsets} = ...

    setName;
    elseif all(ismember(subgroupGlobElems, arr1))
        subgroupName = setName2MtrlAndGlobalElemNrsMap{k,i};
        fprintf(['Subgroup \{name=%s\} is a subset of ', ...
            set\{name=%s\}\n'], ...
            subgroupName,subName);
    elseif any(ismember(arr1,subgroupGlobElems))
        % Test printing!
        D = intersect(arr1,subgroupGlobElems);
        subgroupName = setName2MtrlAndGlobalElemNrsMap{k,i};
        fprintf(['Subgroup \{name=%s\} intersects ', ...
            'with set \{name=%s\} by elements \%s\n'], ...
            subgroupName,subName,num2str(D, '%d,'));
    end
end
end
else
    error(['The set/material \{name=%s\} is not defined in ', ...
        'the Abaqus input file!'],subgroupName);
end
end

% all subgroups have been processed
newGrNr2GlobElemNrsMap{subgroupInd} = unique(newGroupGlobElemNrs);

ind = find(cellfun(@GeneralUtilities.isnonempty,grNr2GrNameMap(i,:)));
if any(ind)
    GroupNames = cell2mat(grNr2GrNameMap(i,ind));
else
    GroupNames = '';
end

ind = find(cellfun(@GeneralUtilities.isnonempty,grNr2SetNameMap(i,:))); % for the test only!
if any(ind)
    SetNames = cell2mat(grNr2SetNameMap(i,ind)); % for the test only!
else
    SetNames = '';
end

D = setxor(currentGrGlobElemNrs,newGroupGlobElemNrs);
if ~all(ismember(CurrentGroupGlobElemNrs,NewGroupGlobElemNrs))
    %D = setdiff(CurrentGroupGlobElemNrs,NewGroupGlobElemNrs);
    if ~isempty(D)
        D = setdiff(currentGrGlobElemNrs,newGroupGlobElemNrs);
        fprintf('FAIL. Group \{name=%s\} is not a union of sets \%s\n', ...
            GroupNames,SetNames,num2str(D, '%d,'));
        fail = true;
    else
        if DEBUG
            fprintf('OK. Group \{name=%s\} is a union of sets \%s\ and materials \{\},\n', ...
                GroupNames,SetNames);
        end
    end
end

% Find the following two mappings
% 1. local to global node number mapping
% 2. local to global element number mapping
% and total number of nodes and elements in the model.
function [instLocal2GlobalNodeNrsMap, nrOfNodes, ...
  instLocal2GlobalElemNrsMap, nrOfElems] = ...
  findLocal2GlobalNodeAndElemMaps(datFilName)
import UtilityPkg.*
import AbaqusPkg.*

nrOfNodes = 0;
nrOfElems = 0;

instLocal2GlobalNodeNrsMap = cell(1,2);
instLocal2GlobalElemNrsMap = cell(1,2);

% Instance name, global element numbers

% % Read Abaqus " . dat " file and build a mapping between the local and
% the global element numbers

Local2GlobalPattern = '
  (LOCAL)\s+TO\s+GLOBAL\s+NODE\s+AND\s+ELEMENT\s+MAPS\n
NodeNodePattern = '\s*(node)\s+node\n
ElemElemPattern = '\s*(element)\s+element\n
InstNameNumNumPattern = ...
  '(\?<name>[a-zA-Z0-9-_.]+)\s+(\d+)\s+(\d+)\s+\s*';

Tokens1 = cell(1);
Tokens1{1} = 'LOCAL';
Tokens1{2} = 'node';
Tokens1{3} = 'element';

% We use the fact that in each instance
% local element numbers start with 1 and
% are written in the " . dat " file in the
% sequential order.
% InstLocal2GlobalElemNrsMap is probably a
% better name

fid = fopen(datFilName);
tline = fgetl(fid);
pattern = Local2GlobalPattern;

while ischar(tline) % if tline is array of characters
  [mat, tok] = regexp(tline, pattern, 'match', 'tokens');
  if ~isempty(tok)
    index = strmatch(tok{1}{1}, Tokens1, 'exact');
    switch index
    case 1 % Reading local to global node and element maps line
      pattern = NodeNodePattern;
      [instLocal2GlobalNodeNrsMap, nrOfNodes] = ...
      ReadLocal2GlobalMap(fid, InstNameNumNumPattern);
      pattern = ElemElemPattern;
      case 2 % Reading node node line
      [instLocal2GlobalNodeNrsMap, nrOfNodes] = ...
      ReadLocal2GlobalMap(fid, InstNameNumNumPattern);
      case 3 % Reading element element line
      end;
    end;
  end;
tline = fgetl(fid);
end;
fclose(fid);
function [SubgroupGlobElems,SubgroupIndex]=FindSubgroupGlobElemNbs (... 
    SubgroupName ,NameDelimiter ,CellArray) 
import UtilityPkg.* 
% the subgroup name is a set name!!
AllSetNames = regexp(CellArray(:,1),NameDelimiter, 'split'); 
strn = strtrim(AllSetNames);
SubgroupGlobElems = [];
SubgroupIndex = [];
% The following part of the code may be done more efficient! 
for m=1:size(strn,1) 
    [row,col]=GeneralUtilities.FindCells(...
        SubgroupName,strn{m,1});
    if ~isempty(row) && ~isnan(row)
        SubgroupIndex = m;
        SubgroupGlobElems = CellArray{SubgroupIndex,2};
    end 
end 
end 

function [tline, instName, instTranslationVector, ...
    instRotAngle, instRotAxisPnt1, instRotAxisPnt2]=readInstanceDef(tline,fid,...
    fidNonUpd, tokens) 
import UtilityPkg.* 
instTranslationVector = [];
instRotAngle = [];
instRotAxisPnt1 = [];
instRotAxisPnt2 = [];
fprintf(fidNonUpd, '%s\n',tline);
if ~isempty(data) &~ isempty(data{1})
    fprintf(fidNonUpd, '%s\n',tline);
    instTranslationVector = [data{1} data{2} data{3}];
end 
% read rotation data if it is defined
% read translation
data = textscan(tline, '%f64 %f64 %f64 %f64');
if ~isempty(data) &~ isempty(data{1})
    fprintf(fidNonUpd, '%s\n',tline);
    instRotAxisPnt1 = [data{1} data{2} data{3}];
    instRotAxisPnt2 = [data{4} data{5} data{6}];
    instRotAngle = data{7};
tline = fgetl(fid);
end 
end 

function [tline,globNodeInd,globCoords]=...
    readNodeCoords(tline,fid,fidNonUpd,...
    instName,... 
    instLocal2GlobalNodeNrsMap,...
    instTranslationVector,...
    instRotAngle,...
    instRotAxisPnt1,instRotAxisPnt2) 
import UtilityPkg.*
globNodeInd = [];
globCoords = [];
fprintf(fidNonUpd, '%s\n',tline);
tline = fgetl(fid);
str = regexp(tline,',' , 'split');
strn = strtrim(str);
um = str2double(strn{1});
while ~isempty(num) && ~isnan(num)
    fprintf(fidNonUpd, '%s\n',tline);
    [instInd, ] = GeneralUtilities.FindCells(...
        instName, instLocal2GlobalNodeNrsMap(:,1));
function [line, N0fElemsInInst, GlobNodeInd, GlobNodeNrs] = ...  
readElemNodeNrs(tline, ...  
fid,fidNonUpd, ...  
InstanceName, ...  
InstLocal2GlobalNodeNrsMap, ...  
InstLocal2GlobalNodeNrsMap)

import UtilityPkg.*

GlobElemInd = { ];
GlobNodeNrs = cell(1,1);
N0fElemsInInst = 0;
fprintf(fidNonUpd, '%s\n', tline);
line = fgetl(fid);
str = regexp(tline, '.', ',','split');
str = strtrim(str);  
num = str2double(str{1});
end
end

% Rodrigues' rotation formula
rotAngRad = instRotAngle*180*pi;
K = instRotAxisPnt2 - instRotAxisPnt1;
normK = norm(K);
if normK>0
  k = K/norm(K);
v = globNodeCoords - instRotAxisPnt1;

vRot = k*cos(rotAngRad) + ...  
cross(k,v)*sin(rotAngRad) + ...  
k*dot(k,v)*(1-cos(rotAngRad));
globNodeCoords = vRot + instRotAxisPnt1;
end
end
globNodeCoords = [globNodeInd, instLocal2GlobalNodeNrsMap{instInd,2}(num)];
str2double(str{3}) ...
str2double(str{4})));
if ~isempty(instTranslationVector)
globNodeCoords = globNodeCoords + ...  
instTranslationVector;
if ~isempty(instRotAngle)
  rotAngRad = instRotAngle*180*pi;
  K = instRotAxisPnt2 - instRotAxisPnt1;
  normK = norm(K);
  if normK>0
    k = K/norm(K);
    v = globNodeCoords - instRotAxisPnt1;
    vRot = k*cos(rotAngRad) + ...  
cross(k,v)*sin(rotAngRad) + ...  
k*dot(k,v)*(1-cos(rotAngRad));
    globNodeCoords = vRot + instRotAxisPnt1;
  end
end
globCoords = [globCoords; globNodeCoords];
tline = fgetl(fid);
str = regexp(tline, '.', '.', 'split');
str = strtrim(str);  
num = str2double(str{1});
end; glBegin

function [line, InstanceNameTmp, LocalSetName, GlobElems] = ...  
readElsetElemNrs(tline, fid,fidNonUpd, CellArray, InstanceName, ...  
InstLocal2GlobalElemNrsMap)

import UtilityPkg.*
fprintf(fidNonUpd, '%s\n', tline);
% Profile is a part of section comment, for example check the following
% comment: ** Section: along–l_1 Profile: along–1
[row.col] = GeneralUtilities.FindCellsi(’Profile’,CellArray);
if isempty(col)
    LocalSetName = CellArray{col(end)+1};
else
    LocalSetName = CellArray{col+1};
end
[row.col] = GeneralUtilities.FindCellsi(’instance’,CellArray);
if isempty(col)
    InstanceNameTmp = upper(CellArray{col+1});
else
    InstanceNameTmp = InstanceName;
end
tline = fgetl(fid);  % element numbers
[row.col] = GeneralUtilities.FindCellsi(’generate’,CellArray);
if isempty(col)
    Elset = fprintf(fidNonUpd,’%s
’,tline);
else
    str = regexp(tline,’\d+’,’match’);
    tline = GeneralUtilities.cell2num(str);
end
tline = fgetl(fid);
Elems = union(Elems,GeneralUtilities.cell2num(strn));
Elems = Elens(1);
tline = fgetl(fid);
Elems = Elens(1);
tline = fgetl(fid);
while isempty(num) && ~isnan(num)  
    fprintf(fidNonUpd,’%s
’,tline);
    Elems = union(Elems,GeneralUtilities.cell2num(strn));
    Elems = Elens(1);
    tline = fgetl(fid);
    str = regexp(tline,’\s+’,’match’);
    strn = strtrim(str);
    num = str2double(strn{1});
end
% InstIndex, col = GeneralUtilities.FindCells(  
%   InstanceNameTmp, InstLocal2GlobalElemNrsMap(:,1));
[InstIndex, col] = findstrmp(  
    InstLocal2GlobalElemNrsMap(:,1), InstanceNameTmp);
GlobElems = InstLocal2GlobalElemNrsMap{InstIndex,2}(Elems);
end

function [SectionComment,LocalSetName,MaterialName,UpdatingStructure] =  
    readSectionDef(tline,fid,...
    fidNonUpd, UpdPartInd, CellArray,...
    InstanceName, UpdatingStructure, SectionComment)
    import UtilityPkg.*

% Profile is a part of section comment, for example check the following
% comment: ** Section: along–l_1 Profile: along–1
[row.col] = GeneralUtilities.FindCellsi(’Profile’,CellArray);
if isempty(col)
    SectionComment = tline;
    LocalSetName = [ ];
    MaterialName = [ ];
else
    [’,col]=GeneralUtilities.FindCellsi(  
        ’material’,CellArray);
    if isempty(col)
        MaterialName = CellArray{col+1};
    else
        [’,col]=GeneralUtilities.FindCellsi(  
        ’else’,CellArray);
    end
    LocalSetName = CellArray{col+1};
else
    if isempty(SectionComment)
        % TODO: check if it is necessary!
        fprintf(fidNonUpd,’%s
’,SectionComment);
    end
else
    fprintf(fidNonUpd,’%s
’,tline);
end
else
    % implicit material definition for
    % section
    % FIXME: REMOVE
% Probably the following code will work
% even for shell general section if
1281 NrOfLines = AbaqusPkg.GeneralSection.getNrOfLines();
1282 str = tline;
1283 for k=1:NrOfLines-1 % 1 line of definition are already read
1284 str = sprintf('%s
%s',str,str1get(fid));
1285 end
1286 Section=AbaqusPkg.GeneralSection.fromString(...
1287 str,InstanceName);
1288 setComment(Section,SectionComment);
1289 SectionComment = [];
1290 LocalSetName = getElsetName(Section);
1291 MaterialName = getName(Section);
1292 UpdatingStructure = addSection(...
1293 UpdatingStructure,UpdPartInd,Section);
1294 end
1295 end
1296
1297 % Output:
1298 % GlobNodes -- column vector of global node numbers corresponding to spring
1299 function [tline,UpdatingStructure,Spring,GlobNodes,GlobElems,...
1300 SpringDof]=readSpringDef(tline,fid,...
1301 UpdPartInd,InstanceName,...
1302 InstLocal2GlobalElemNrsMap,...
1303 InstLocal2GlobalNodeNrsMap,...
1304 UpdatingStructure)
1305 import UtilityPkg.
1306 NrOfLines = AbaqusPkg.Spring.getNrOfLines();
1307 str = tline;
1308 for k=1:NrOfLines-1 % 1 line of definition is already read
1309 str = sprintf('%s
%s',str,str1get(fid));
1310 end
1311 tline = str1get(fid);
1312 str = regexp(tline, ',','split');
1313 strn = strtrim(str);
1314 ElemNr = str2double(strn{1});
1315 GlobNodes = [];
1316 GlobElems = [];
1317 ElemNodeNrList = [];
1318 SpringDof = getDof(Spring);
1322 while ~isempty(ElemNr) && ~isnan(ElemNr)
1323 NodeNr = str2double(strn{2});
1324 NewNode = dnode(...
1325 AbaqusPkg.SpringElemNodeData(ElemNr,NodeNr));
1326 if ~isempty(ElemNodeNrList)
1327 insertAfter(NewNode,ElemNodeNrList);
1328 else
1329 ElemNodeNrList = NewNode;
1330 end
1332 if ~isempty(InstanceName)
1333 [InstIndex,col]=GeneralUtilities.FindCells(...
1334 InstanceName,InstLocal2GlobalElemNrsMap(:,1));
1335 GlobElemNr = InstLocal2GlobalElemNrsMap{InstIndex,2}(ElemNr);
1336 [InstIndex,col]=GeneralUtilities.FindCells(...
1337 InstanceName,InstLocal2GlobalNodeNrsMap(:,1));
1338 GlobNodeNr = InstLocal2GlobalNodeNrsMap{InstIndex,2}(NodeNr);
1339 else
1340 GlobElemNr = ElemNr;
1341 GlobNodeNr = NodeNr;
1342 end
1344 GlobNodes = union(GlobNodes,GlobNodeNr);
1346 tline = str1get(fid);
1347 str = regexp(tline, ',','split');
1348 strn = strtrim(str);
1349 ElemNr = str2double(strn{1});
end
GlobElems = GlobElems(:,);
GlobNodes = GlobNodes(:,);
setElemNodeNrList(Spring,ElemNodeNrList);
UpdatingStructure = addSpring(UpdatingStructure,...
UpdPartInd,Spring);
end

function [tline,fidNonUpd,UpdPartInd,NonUpdFileNames,Material,...
MaterialName] = readMaterialDef(fid,fidNonUpd,CellArray,...
UpdPartInd,...
UpdatingStructure,...
NonUpdFileNames,...
NonUpdFileNameCommon,...
Material)
import AbaqusPkg.*

MaterialPattern = 's*(Material),s*name=(\[a-zA-Z0-9_]+)';
ConductivityPattern = 's*(Conductivity)';
DampingPattern = 's*(Damping)';
DensityPattern = 's*(Density)';
ElasticPattern = 's*(Elastic)';
ExpansionPattern = 's*(Expansion)';

% Number patterns
%NumPattern = ['-\+[0-9]*\[[0-9]*\[[0-9]+']
%ExpNumPattern = ['-\+[0-9]*\d+\d*\[[eE]\+-\[[0-9]+']

pattern = [MaterialPattern ' ' DensityPattern ' ']

ElasticPattern ' ' ConductivityPattern ' ' ...
DampingPattern ' ' ExpansionPattern];

Tokens = cell(1);
Tokens{1} = 'Density';
Tokens{2} = 'Elastic';
Tokens{3} = 'Conductivity';
Tokens{4} = 'Damping';
Tokens{5} = 'Expansion';

[tline,fidNonUpd,UpdPartInd,NonUpdFileNames,Material,...
MaterialName] = readMaterialDef(fid,...
 fidNonUpd,CellArray,...
 Material,...
 UpdPartInd,UpdatingStructure,...
 NonUpdFileNames,...
 NonUpdFileNameCommon);
[mat,tok] = regexp(tline,pattern,'match','tokens','ignorecase');

while ~isempty(tok)
disp(tline)
index = find(strcmpi(tok{1}{1},Tokens)); % case insensitive!
if ~isempty(index)
switch index
  case 1 % Reading material density
    tline = fgets(fid);
    str = regexp(tline,'[,]','split');
    str = str trim(str);
    num = str2double(str{1});
    if ~isempty(num) && ~isnan(num)
      setDensity(Material,num);
    end
  case 2 % Reading material elastic properties
    tline = fgets(fid);
    str = regexp(tline,'[,]','split');
    str = str trim(str);
    num = str2double(str{1});
    if ~isempty(num) && ~isnan(num)
      setElastModulus(Material,num);
    end
  case 3 % Reading material conductivity
    tline = fgets(fid);
end
```
tline = fg1t(fid);
str = regexp(tline,’,:,’,’split’);
strn = strtrim(str);
num = str2double(strn{1});
if ~isempty(num) && ~isnan(num)
    setDamping(Material,num);
tline = fg1t(fid);
end

% Reading material damping

case 4
    tline = fg1t(fid);
    str = regexp(tline,’,:,’,’split’);
    strn = strtrim(str);
    num = str2double(strn{1});
    if ~isempty(num) && ~isnan(num)
        setDamping(Material,num);
tline = fg1t(fid);
end

% Reading material expansion

case 5
    tline = fg1t(fid);
    str = regexp(tline,’,:,’,’split’);
    strn = strtrim(str);
    num = str2double(strn{1});
    if ~isempty(num) && ~isnan(num)
        setExpansion(Material,num);
tline = fg1t(fid);
end

% store previous material definition
addMaterial(UpdatingStructure,UpdPartInd,...
Material);
end

if isMtrlPartEmpty(UpdatingStructure)
    UpdPartInd = UpdPartInd + 1;
    fclose(fidNonUpd);
    NonUpdFileNames{UpdPartInd,1} = sprintf(’%s_PART%d’,...
    NonUpdFileNameCommon,UpdPartInd);
    fidNonUpd = fopen(NonUpdFileNames{UpdPartInd,1},’w’);
end

function [NrOfModes,tline] = readTotNrOfModes(tline,...
    fid,fidNonUpd)
str = regexp(tline,’,:,’,’split’);
strn = strtrim(str);
um = str2double(strn{1});
if ~isempty(num) && ~isnan(num)
    NrOfModes = num;
    fprintf(fidNonUpd, ’%s\n’,tline);
tline = fg1t(fid);
end

function Dofs = Local2GlobalInd(NodesInElem,dofNrs,NrOfDofsPerNode)
LocalNofDofs = length(dofNrs);
nofNodesInElem = length(NodesInElem);

offset = LocalNofDofs*(NodesInElem-1);
tmp = offsets + repmat(dofNrs(:,1),1,nofNodesInElem);

% LocalNofDofs = length(dofNrs);
% offset = LocalNofDofs*(NodesInElem-1);
% tmp = offsets + repmat(dofNrs(:,1),1,nofNodesInElem);
% Dofs = tmp(:,);

function testGroupsForIntersections(grGlobElemMap, grNr2GrNameMap, ... DEBUG)
import UtilityPkg.*
if DEBUG
    disp('**************************************************************************
    *** GROUP INTERSECTION TEST ***
    **************************************************************************
    end;
fail = false;
[nrOfGroups, ] = size(grGlobElemMap);
for i=1:nrOfGroups
    grElems = grGlobElemMap{i,1};
    for j=1:nrOfGroups
        otherGrElems = grGlobElemMap{j,1};
        if i=j
            D = intersect(grElems, otherGrElems);
            if isempty(D)
                ind = find(cellfun(@(GeneralUtilities isempty, ...
                    grNr2GrNameMap{i,:}));
                grNames = '';
                otherGrNames = '';
                if any(ind)
                    grNames = cell2mat(grNr2GrNameMap{i,ind});
                end
                ind = find(cellfun(@(GeneralUtilities isempty, ...
                    grNr2GrNameMap{j,:})); % for the test only!
                if any(ind)
                    otherGrNames = cell2mat(grNr2GrNameMap{j,ind}); % for the test only!
                end
                fprintf(['Group {index= %d, names= %s} intersects group ', ...
                    {index= %d, names= %s} by elements {%s}\n']...,
                    i,grNames,j,otherGrNames,num2str(D, '%d\%d'));
                fail = true;
            end
        end
    end
end
end

if ~fail
    if DEBUG
        disp('OK. Any two groups do not intersect each other.\n    end
    else
        disp('Group definition has failed to pass the group intersection test!\n    error(''Group definition has failed to pass the group intersection test!'');
end
end

J.2.3 The GeneralSection class
Source code
```matlab
classdef GeneralSection < AbaqusPkg.UpdElastElement
% Example: for Lngforsen bridge
% BFAM GENERAL SECTION, ELSET=<element set name>, SECTION=<section type>
% <cross-section dimensions> or <section engineering properties>
% <n_1^1>,<n_1^2>,<n_1^3>
% Young's modulus (E),<torsional shear modulus (G)>
% By Abaqus 6.13 manual 6.1 Beam cross--section geometry
% <cross-section dimensions> or <section engineering properties> = Area, I_1^1, I_1^2, I_1^2, \leftrightarrow
torsional rigidity
% <n_1^1>,<n_1^2>,<n_1^3>
% Young's modulus (E),<torsional shear modulus (G)>
% By Abaqus 6.13 manual 6.1 Beam cross--section geometry
% Use textscan to read a string of comma-separated doubles
% Ex. C = textscan(str,,'%f, %f, %f, %f')
% Then, use C{i}
% Is this the same for Shell General Section if there is such?

properties
%SectionName % string
ProfileName % string
comment % string
elsetName % string
density % double value
sectionDesc % string
sectionEngProperties % string with 1–5 doubles
sectionNnumbers % string with 3 doubles
elastModulus % double with exponent
torsionalShearModulus % double with exponent
materialMatlabGenName % name of material assigned by Matlab
% for internal usage only

end

methods (Access = public)
function obj = GeneralSection(elsetName)
%obj.SectionName = SectionName;
%obj.ProfileName = ProfileName;
obj.elsetName = elsetName;
end
function obj = setComment(obj,comment)
obj.comment = comment;
end
function obj = setPoisson(obj,poisson)
obj.poiisson = poisson;
end
function obj = setDensity(obj,density)
obj.density = density;
end
function obj = setSectionDesc(obj,sectionDesc)
obj.sectionDesc = sectionDesc;
end
function obj = setSectionEngProperties(obj,sectionEngProperties)
obj.sectionEngProperties = sectionEngProperties;
end
function obj = setSectionNnumbers(obj,sectionNnumbers)
obj.sectionNnumbers = sectionNnumbers;
end
function obj = setElastModulus(obj,elastModulus)
obj.eastModulus = elastModulus;
end
function obj = setTorsionalShearModulus(obj,torsionalShearModulus)
obj.torsionalShearModulus = torsionalShearModulus;
end
function obj = setName(obj,materialMatlabGenName)
obj.materialMatlabGenName = materialMatlabGenName;
end
function value = getElsetName(obj)
value = obj.elsetName;
end
function value = getPoisson(obj)
end
```
function value = getOrientations(obj)
    value = obj.orientations;
end

function value = getDensity(obj)
    value = obj.density;
end

function value = getSectionDesc(obj)
    value = obj.sectionDesc;
end

function value = getComment(obj)
    value = obj.comment;
end

function value = getSectionEngProperties(obj)
    value = obj.sectionEngProperties;
end

function value = getSectionNnumbers(obj)
    value = obj.sectionNnumbers;
end

function value = getElastModulus(obj)
    value = obj.elastModulus;
end

function value = getTorsionalShearModulus(obj)
    value = obj.torsionalShearModulus;
end

function value = getName(obj)
    value = obj.materialMatlabGenName;
end

function value = toString(obj)
% Example
% ** Section: U2U3  Profile: U2U3  Elset=Set-U2U3_9, poisson = 0.3, density=7800., section=--
% GENERAL
% 0.02436, 0.0002704, 0., 0.0007679, 3.62e-06
% 0..0..1.
% 2.1e+11, 8.08e+10

%value = sprintf('%s
', [SectionNameStr obj.SectionName ' ' ...
%    ProfileNameStr obj.ProfileName ]); 
value = '%s
';
if ~isempty(obj.comment)
    value = sprintf('%s
', obj.comment);
end
value = sprintf('%s', [value ElsetStr obj.elsetName]);
if ~isempty(obj.poisson)
    value = sprintf('%s, %s%g ', value, PoissonStr, obj.poisson);
end
if ~isempty(obj.density)
    value = sprintf('%s, %s%g ', value, DensityStr, obj.density);
end
if ~isempty(obj.sectionDesc)
    value = sprintf('%s
', obj.sectionDesc);
end
if ~isempty(obj.sectionEngProperties)
    value = sprintf('%s
', obj.sectionEngProperties);
end
if ~isempty(obj.sectionNnumbers)
    value = sprintf('%s
', obj.sectionNnumbers);
end
if ~isempty(obj.elastModulus) & & ~isempty(obj.torsionalShearModulus)
    value = sprintf('%s
', [value, obj.elastModulus, obj.←
    torsionalShearModulus]);
end
value = sprintf('%s
', value);
end
methods (Static)

function obj = fromString(str, InstanceName)
  %p1 = ['
  p2 = ['
  p3 = ['
  p4 = ['
  p5 = ['

obj = AbaqusPkg.GeneralSection(c.ElsetName);
setPoisson(obj, str2double(c.Poisson));
setDensity(obj, str2double(c.Density));
setSectionDesc(obj, c.SectionDesc);
secEngProps = c.prop1;
if ~isempty(c.prop2)
  secEngProps = sprintf('%s , %g'

function NrOfLines = getNrOfLines()
NrOfLines = 4;
end
end

J.2.4 The Material class

Source code

classdef Material < AbaqusPkg.UpElastElement
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se

properties
  name % string
  conductivity % double value
  damping % double value
  elastModulus % double value
  density % double value
  poisson % double value
end

expansion % double value

methods (Access = public)
  function obj = Material(name)
    obj.name = name;
  end
  function obj = setName(name)
    obj.name = name;
  end
  function obj = setConductivity(obj, conductivity)
    obj.conductivity = conductivity;
  end
  function obj = setDamping(obj, damping)
    obj.damping = damping;
  end
  function obj = setDensity(obj, density)
    obj.density = density;
  end
  function obj = setElastModulus(obj, elastModulus)
    obj.elastModulus = elastModulus;
  end
  function obj = setPoisson(obj, poisson)
    obj.poisson = poisson;
  end
  function obj = setExpansion(obj, expansion)
    obj.expansion = expansion;
  end
  function value = getName(obj)
    value = obj.name;
  end
  function value = getConductivity(obj)
    value = obj.conductivity;
  end
  function value = getDamping(obj)
    value = obj.damping;
  end
  function value = getDensity(obj)
    value = obj.density;
  end
  function value = getPoisson(obj)
    value = obj.poisson;
  end
  function value = getElastModulus(obj)
    value = obj.elastModulus;
  end
  function value = getExpansion(obj)
    value = obj.expansion;
  end
  function value = toString(obj)
    value = obj.name;
  end

% Example 1:
% Material., name=concrete
% density =2500.000000,
% Elastic =3.59e+10, 0.200000

% Example 2:
% Material., name=concrete
% Conductivity =0. ,
% Damping =
% Density =2500. ,
% Elastic =37e+9, 0.2
% Expansion =1e–05. 

 /////////// Tag definitions /////////// 
 MaterialName = ' ∗Material, name=';
 DensityStr = ' ∗Density';
 ElasticStr = ' ∗Elastic';
ConductivityStr = 'sConductivity';
DampingStr = 'sDamping';
ExpansionStr = 'sExpansion';

value = sprintf('%s', [MaterialName obj.name]);
if isempty(obj.conductivity)
  value = sprintf('%s
%g
', value, ConductivityStr, obj.conductivity);
end
if isempty(obj.damping)
  value = sprintf('%s
%g
', value, DampingStr, obj.damping);
end
if isempty(obj.density)
  value = sprintf('%s
%g
', value, DensityStr, obj.density);
end
if isempty(obj.elastModulus) &
  isempty(obj.poisson)
  value = sprintf('%s
%g
%g
', value, ElasticStr, obj.elastModulus, obj.poisson);
end
if isempty(obj.expansion)
  value = sprintf('%s
%g
', value, ExpansionStr, obj.expansion);
end
value = sprintf('%s
', value);

methods (Static)
function obj = fromString(str)
  p1 = 'sMaterial
  p2 = 'sConductivity
  p3 = 'sDamping
  p4 = 'sDensity
  p5 = 'sElastModulus
  p6 = 'sExpansion
  p = [p1 's' p2 's' p3 's' p4 's' p5 's' p6];
  c = regexp(str, p, 'names');
  if isempty(c.Conductivity)
    obj.setConductivity(str2double(c.Conductivity));
  end
  if isempty(c.Damping)
    obj.setDamping(str2double(c.Damping));
  end
  if isempty(c.Expansion)
    obj.setExpansion(str2double(c.Expansion));
  end
  obj.setDensity(str2double(c.Density));
  obj.setElastModulus(str2double(c.ElastModulus));
  obj.setPoisson(str2double(c.Poisson));

function NrOfLines = getNrOfLines()
NrOfLines = []; end
end

J.2.5 The Spring class

Source code
methods
function obj = Spring(elset)
    obj.elset = elset;
end
function value = getName(obj)
    value = obj.name;
end
function obj = setName(obj, name)
    obj.name = name;
end
function value = getElsetName(obj)
    value = obj.elset;
end
function obj = setElsetName(obj, elset)
    obj.elset = elset;
end
function setDof(obj, dof)
    obj.dof = dof;
end
function value = getDof(obj)
    value = obj.dof;
end
function setElastModulus(obj, elastModulus)
    obj.elastModulus = elastModulus;
end
function value = getElastModulus(obj)
    value = obj.elastModulus;
end
function setElementType(obj, elemType)
    obj.elemType = elemType;
end
function value = getElementType(obj)
    value = obj.elemType;
end
function setElemNodeNrList(obj, elemNodeNrList)
    obj.elemNodeNrList = elemNodeNrList;
end
function value = getElemNodeNrList(obj)
    value = obj.elemNodeNrList;
end
function value = toString(obj)
    % Example:
    % Spring, elset=arch-4-spring_1
    % 2.5e+11
    % Element, type=Spring1, elset=arch-4-spring_1
    % 4980, 843
    value = sprintf('%s', [SpringElset obj.elset]);
    value = sprintf('%s
%d', value, obj.dof);
    value = sprintf('%s
%g
%4', value, obj.elastModulus);
    value = sprintf('%s
%4980, 843
%Element, type=Spring1, elset=arch-4-spring_1
%2.5e+11', value, obj.elset);

    node = obj.elemNodeNrList;
    while (~isempty(node))
        data = node.Data;
        value = sprintf('%s
%4', value, toString(data));
        node = node.Next;
    end
end
J.2.5.1 The method

J.2.6 The SpringElemNodeData class

Source code
J.2.7 The UpdatingStructure class

Source code

```matlab
classdef UpdatingStructure < handle
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se

    % properties (Access = private)
    properties (Access = private)
        springListHeadCellArray
        mtrlListHeadCellArray
        sectionListHeadCellArray  % material properties for beam (shell?) general section
        % are defined explicitly
        springListTailCellArray
        mtrlListTailCellArray
        sectionListTailCellArray
    end

    methods

        function obj = UpdatingStructure()
            obj.mtrlListHeadCellArray = cell(1,1);
            obj.springListHeadCellArray = cell(1,1);
            obj.sectionListHeadCellArray = cell(1,1);
            
            % to insert elements at the end of the list
            obj.mtrlListTailCellArray = cell(1,1);
            obj.springListTailCellArray = cell(1,1);
            obj.sectionListTailCellArray = cell(1,1);
        end

        function update(obj, P, paramInd2GrNameMap)
            updateCellArray(obj, obj.mtrlListHeadCellArray, P, paramInd2GrNameMap);
            updateCellArray(obj, obj.springListHeadCellArray, P, paramInd2GrNameMap);
            updateCellArray(obj, obj.sectionListHeadCellArray, P, paramInd2GrNameMap);
        end

        function value = isMtrlPartEmpty(obj)
            value = all(cellfun(@isempty, obj.mtrlListHeadCellArray));
        end

        function value = mtrlList2String(obj, index)
            value = [];
            %Materials = '** MATERIALS';
            %Stars = '** ';
            
            if size(obj.mtrlListHeadCellArray,2)>index && ...  
                isempty(obj.mtrlListHeadCellArray{index})
                value = sprintf('%s\n%s\n%s\n', Stars, Materials, Stars);
            node = obj.mtrlListHeadCellArray{index};
            while (~isempty(node))
                data = node.Data;
                value = sprintf('%s\n', [value toString(data)]);
                node = node.Next;
            end
            %value = sprintf('%s\n', [value Stars]);

        end

        function value = springList2String(obj, index)
            value = [];
            if size(obj.springListHeadCellArray,2)>index && ...  
                isempty(obj.springListHeadCellArray{index})
                node = obj.springListHeadCellArray{index};
            while (~isempty(node))
                data = node.Data;
                value = sprintf('%s\n', [value toString(data)]);
        end
```
function value = sectionList2String(obj, index)
    value = []; 
    if size(obj.sectionListHeadCellArray,2)>=index & & ... 
        node = obj.sectionListHeadCellArray{index}; 
        while (~isempty(node)) 
            data = node.Data; 
            value = sprintf('%s', [value toString(data)]); 
            node = node.Next; 
        end 
    end 
end 

function obj = addSpring(obj, index, spring)
    import UtilityPkg.dlnode
    NewNode = dlnode(spring); 
    if size(obj.springListHeadCellArray,2)<index | | ... 
        isempty(obj.springListHeadCellArray{index}) 
        obj.springListHeadCellArray{index} = NewNode; 
        insertAfter(NewNode, obj.springListTailCellArray{index}); 
    end 
    obj.springListTailCellArray{index} = NewNode; 
end 

function obj = addMaterial(obj, index, material)
    import UtilityPkg.dlnode
    NewNode = dlnode(material); 
    if size(obj.mtrlListHeadCellArray,2)<index | | ... 
        isempty(obj.mtrlListHeadCellArray{index}) 
        obj.mtrlListHeadCellArray{index} = NewNode; 
        insertAfter(NewNode, obj.mtrlListTailCellArray{index}); 
    end 
    obj.mtrlListTailCellArray{index} = NewNode; 
end 

function obj = addSection(obj, index, section)
    import UtilityPkg.dlnode
    NewNode = dlnode(section); 
    if size(obj.sectionListHeadCellArray,2)<index | | ... 
        isempty(obj.sectionListHeadCellArray{index}) 
        obj.sectionListHeadCellArray{index} = NewNode; 
        insertAfter(NewNode, obj.sectionListTailCellArray{index}); 
    end 
    obj.sectionListTailCellArray{index} = NewNode; 
end 

function value = isMaterialName(obj, name)
    value = false; 
    for i=1:size(obj.mtrlListHeadCellArray,2)
        node = obj.mtrlListHeadCellArray{i}; 
        while (~isempty(node)) 
            data = node.Data; 
            if strcmp(getName(data), name)
                value = true; 
                break; 
            end 
            node = node.Next; 
        end 
    end 
end 

function value = isSpringName(obj, name)
    value = false; 
    for i=1:size(obj.springListHeadCellArray,2)
        node = obj.springListHeadCellArray{i}; 
    end 
end
while (~isempty(node))
    data = node.Data;
    if strcmp(getElsetName(data), name)
        value = true;
        break;
    end
    node = node.Next;
end

function value = isSectionName(obj, name)
    value = false;
    for i=1:size(obj.sectionListHeadCellArray,2)
        node = obj.sectionListHeadCellArray{i};
        while (~isempty(node))
            data = node.Data;
            if strcmp(getName(data), name)
                value = true;
                break;
            end
            node = node.Next;
        end
    end
end

function value = isempty(obj, index)
    value = isempty(obj.mtrlListHeadCellArray{index}) | ...
            isempty(obj.springListHeadCellArray{index}) | ...
            isempty(obj.sectionListHeadCellArray{index});
end

% All spring lists are composed into one list!
% TODO Is this function needed?
function list = getSpringList(obj)
    list = [];
    listTail = [];
    n = size(obj.springListHeadCellArray,2);
    for i=1:n
        if ~isempty(obj.springListHeadCellArray{i})
            node = obj.springListHeadCellArray{i};
            while (~isempty(node))
                if isempty(list)
                    list = node;
                else
                    insertAfter(node, listTail);
                end
                listTail = node;
                node = node.Next;
            end
        end
    end
end

function updateCellArray(obj, cellArray, P, paramInd2GrNameMap)
    import UtilityPkg.*
    for i=1:size(cellArray,2)
        node = cellArray{i};
        while (~isempty(node))
            data = node.Data;
            [GroupIndex, col] = GeneralUtilities.FindCells(getName(data), ...
                paramInd2GrNameMap);
            if ~isempty(GroupIndex)
                setElastModulus(data, P(GroupIndex));
            end
            node = node.Next;
        end
    end
end
### J.2.8 The UpdElastElement class

**Source code**

```matlab
classdef UpdElastElement < handle
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    methods (Abstract)
        setName(obj, name)
            % Name returned by getName(obj) is used for group definition
            % It can be either material name or spring name for the moment.
            % When it is a material name connected to the Beam General Section
            % definition it is generated by Matlab and used internally in the
            % simulator part of the optimization. The name contains the elset
            % name and the instance name.
        getName(obj)
        setElastModulus(obj, E)
        getElastModulus(obj)
        toString(obj)
    end
end
```

### J.3 ArtemisPkg package

The ArtemisPkg package contains methods for reading and writing modal data created with the ARTeMIS operational modal analysis software.

#### J.3.1 The Utilities class

**Source code**

```matlab
classdef Utilities
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    methods (Static)
        % Read directory with Artemis modal data files.
        %
        % Input:
        % artFilesDir — directory with artemis files
        %
        % Output:
        % artNodeCoords — Nx4 matrix each row of which contains artemis
        % model node number nad x, y and z coordinates
        % of this node in the artemis model coordinate system
        % elemNode — cell array with entries containing nodes numbers
        % for each element
        % freq — eigen frequencies in the increasing order
        % damping — damping for the mode shapes corresponding to freq
        % vector
        % modes — mode shapes corresponding to freq vector. For each node
        % the data in the mode shape comes in the order of x, y and
        % z coordinates of the artemis model
        function [artNodeCoords, elemNode, freq, damping, modes] = ...
            readArtemisModalDataFiles(artFilesDir)
        artFiles = dir([artFilesDir '\*.svs']);
        nrOfArtFiles = size(artFiles,1);
        nodes = 'Nodes';
        linesStr = 'Lines';
```
```matlab
surfacesStr = 'Surfaces';
frequency = 'FREQUENCY';
dampingStr = 'DAMPING';
mode_shape = 'MODE SHAPE';
end_mode_definition = 'END MODE DEFINITION';
ln = length(nodestr);
fn = length(frequency);\ndn = length(dampingStr);
mn = length(mode_shape);
em = length(end_mode_definition);
disp('Reading Artemis files');
for i=1:.nrOfArtFiles
    fid = fopen(fullfile(artFilesDir,'\artFiles(i).name'));
    tline = fgets(fid);
    artNodeCoords = [];  
    % 1st column is Artemis node number,  
    % 2-4 columns Artemis x,y,z coordinates  
    lines = [];  
    surfaces = [];  
    modeShape = [];  
    nrOfNodes = 0;  
    nrOfLines = 0;  
    nrOfSurfaces = 0;
    while ~isempty(tline)
        %disp(tline)
        if length(tline)>=ln & strcmp(tline(1:ln),linesStr)
            tline = fgets(fid);
            str = regexp(tline,'\', '+', '\');
            str = strtrim(str);
            str = regexp(str,'\', 'split');
            num = str2double(str{1});
            while ~isempty(num) & ~isnan(num)
                nrOfLines = nrOfLines + 1;
                lines(nrOfLines,1:2) = ...  
                    str2double(str{1}) ...  
                    str2double(str{2})];
                tline = fgets(fid);
                str = regexp(tline,'\', '+', '\');
                str = strtrim(str);
                str = regexp(str,'\', 'split');
                num = str2double(str{1});
            end;
        end;
        if length(tline)>=nn & strcmp(tline(1:nn),nodesStr)
            tline = fgets(fid);
            while isempty(tline)
                tline = fgets(fid);
            end
            data = textscan(tline, '%d%f64%f64%f64');
            num = data{1};
            while ~isempty(num) & ~isnan(num)
                nrOfNodes = nrOfNodes + 1;
                artNodeCoords(nrOfNodes,1) = num;
                artNodeCoords(nrOfNodes,2:4) = ...  
                    [data{2} data{3} data{4}];
                tline = fgets(fid);
                if ~isempty(tline)
                    data = textscan(tline, '%d%f64%f64%f64');
                    num = data{1};
                else
                    num = [ ];
                end
            end
        end;
        if length(tline)>=sn & strcmp(tline(1:sn),surfacesStr)
            tline = fgets(fid);
            str = regexp(tline,'\', '+', '\');
        end
    end;
end;
```
str = strtrim(str);
strn = regexp(str, [', ',',','split'],'s');
num = str2double(strn);
while ~isempty(num) && ~isnan(num(1))
    nrOfSurfaces = nrOfSurfaces + 1;
surfaces(nrOfSurfaces, 1:length(num)) = num;
tline = fgets(fid);
    str = regexprep(tline, [', ',],' ', 'split');
    str = strtrim(str);
strn = regexp(str, [', ',],'split');
    num = str2double(strn);
end;
if length(tline)>=fn && strcmp(tline(1:fn), frequency)
    % disp(tline);
    tline = fgets(fid);
    temp = strread(tline);
    freq(i) = temp(1);
end;
if length(tline)>=dn && strcmp(tline(1:dn), dampingStr)
    % disp(tline);
    tline = fgets(fid);
    temp = strread(tline);
    damping(i) = temp(1);
end;
if length(tline)>=mn && strcmp(tline(1:mn), mode_shape)
    tline = fgets(fid);
    n = 0;
    while ischar(tline) && length(tline)>=en && ~strcmp(tline(1:en), ←)
        end_mode_definition)
        temp = strread(tline);
        % Artemis: X X−ang Y Y−ang Z Z−ang
        modeShape = [modeShape temp(2:end)];
        %ind=find(MeaNodeNrs==temp(1));
        % find index of temp(1) in the array nodes_art
        % If ~isempty(ind)
        %OMAModeStruct(i).nodes(ind) = temp(1);
        % FEM: X Y Z RotX RotY RotZ
        %[a,b]=ismember([1 2 3], Mea2FemDofsMap(ind,:));
        %[row,col,v] = find(Mea2FemDofsMap(ind,:));
        %v = nonzeros(b);
        %data2(n+1:n+length(v)) = temp(2*v').*cos(temp(2*v'+1));
        %v = [1 2 3]';
        %data2(n+1:n+length(v')) = temp(2*v').*cos(temp(2*v+1));
        %v = find(Mea2FemDofsMap(ind,:));
        %data2(n+1:n+length(v')) = temp(2*v').*cos(temp(2*v'+1));
        %n = n + length(v);
        %end;
        tline = fgets(fid);
        end;
        modes(:,i) = modeShape';
    end;
tline = fgets(fid);
fclose(fid);
elemNode = cell(1,1);
for l=1:length(lines)
    elemNode{l,1} = lines(l,:);
end
% for m=1:length(Surfaces)
%    ElemNodeCoord{1,1-mm} = NodeCoords(Surfaces(mm,:,:));
%end

freq = freq';
[freq,ind] = sort(freq, 'ascend');
damping = damping(ind);
modes = modes(:,ind);
function [artNodeCoords, testModelGraph, freq, damping, modes] = readArtemisModalDataFile(artFile)
	nodes = 'Nodes';
	linesStr = 'Lines';
surfacesStr = 'Surfaces';
frequency = 'FREQUENCY';
dampingStr = 'DAMPING';
mode_shape = 'MODE SHAPE';
begin_mode_definition = 'BEGIN MODE DEFINITION';
end_mode_definition = 'END MODE DEFINITION';
nn = length(nodes);
ln = length(linesStr);
sn = length(surfacesStr);
fn = length(frequency);
dn = length(dampingStr);
mn = length(mode_shape);n = length(begin_mode_definition);
b = length(end_mode_definition);

fid = fopen(artFile);
tline = fgets(fid);
	node = [];
	testModelGraph = cell(1,1);
surfaces = [];
	noOfNodes = 0;
	noOfLines = 0;
	noOfSurfaces = 0;
i = 0;
while ischar(tline)
    if length(tline) >= bn & strcmp(tline(1:bn),begin_mode_definition)
        i = i + 1;
        modeShape = [];
        elseif length(tline) >= ln & strcmp(tline(1:ln), linesStr)
            tline = fgets(fid);
            str = regexp(tline, ' +', 'split');%str = strtrim(str);
            strn = regexp(str, ' ', 'split');
            num = str2double(strn{1});
            while ~isempty(num) & ~isnan(num)
                noOfLines = noOfLines + 1;
                testModelGraph{i, noOfLines} = ...%str2double(strn{1}) ...%str2double(strn{2}) 
                num = str2double(strn{2});
                tline = fgets(fid);
                str = regexp(tline, ' +', 'split');%str = strtrim(str);
                strn = regexp(str, ' ', 'split');
            end
        end
        testModelGraph{i, noOfLines} = ...%str2double(strn{1}) ...%str2double(strn{2})
        tline = fgets(fid);
        str = regexp(tline, ' +', 'split');%str = strtrim(str);
        strn = regexp(str, ' ', 'split');
    end
end

%disp(['Reading Artemis file ', artFile]);
```matlab
num = str2double(strn{1});
end;
elseif length(tline) >= mn && strcmp(tline(1:mn),nodes)
tline = fgets(fid);
while isempty(tline)
tline = fgets(fid);
end
data = textscan(tline, '%d%f64%f64%f64 ');
um = data{1};
while ~isempty(num) && ~isnan(num)
nrOfNodes = nrOfNodes + 1;
artNodeCoords(nrOfNodes,1) = num;
artNodeCoords(nrOfNodes,2:4) = ...
[data{2} data{3} data{4}];
tline = fgets(fid);
if ~isempty(tline)
data = textscan(tline, '%d%f64%f64%f64 ');
um = data{1};
else
num = [];
end;
elseif length(tline) >= sn && strcmp(tline(1:sn),surfacesStr)
tline = fgets(fid);
str = regexprep(tline, ' ', '+', ' ');
str = strtrim(str);
str = regexprep(str, ' ', 'split');
um = str2double(str);
while ~isempty(num) && ~isnan(num(1))
nrOfSurfaces = nrOfSurfaces + 1;
surfaces(nrOfSurfaces,1:length(num)) = num;
tline = fgets(fid);
str = regexprep(tline, ' ', '+', ' ');
str = strtrim(str);
str = regexprep(str, ' ', 'split');
um = str2double(str);
end;
elseif length(tline) >= fn && strcmp(tline(1:fn),frequency)
disp(tline);
tline = fgets(fid);
temp = strread(tline);
if ~isempty(tline)
fn=temp(1);
end;
elseif length(tline) >= dn && strcmp(tline(1:dn),dampingStr)
disp(tline);
tline = fgets(fid);
temp = strread(tline);
damping(i) = temp(1);
elseif length(tline) >= mn && strcmp(tline(1:mn),mode_shape)
tline = fgets(fid);
n = 0;
while ischar(tline) & length(tline) >= en & strcmp(tline(1:en), end_mode_definition)
temp = strread(tline);
if ~isempty(tline)
%d Artemis: X X-ang Y Y-ang Z Z-ang
v = [2 4 6]:
modeShape = [modeShape temp(v).*cos(temp(v+1))];
end
ind=find(MeaNodeNrs==temp(1));
if ~isempty(ind)
OMAModeStruct(ind).nodes(ind) = temp(1);
end
FEM: X Y Z RotX RotY RotZ
%d [a,b]=ismember([1 2 3],Mea2FemDofsMap(ind,:));
[row, col, v] = find(Mea2FemDofsMap(ind,:));
v = nonzeros(v);
data2(n+1:n+length(v')) = temp(2*v').*cos(temp(2*v'+1));
v = [1 2 3]:
data2(n+1:n+length(v')) = temp(2*v').*cos(temp(2*v'+1));
v = find(Mea2FemDofsMap(ind,:));
data2(n+1:n+length(v')) = temp(2*v').*cos(temp(2*v'+1));
end;
```
J.4 MatlabSimPkg package

See Section A.1

J.4.1 The MatlabModalDataSim class

Source code

```matlab
classdef MatlabModalDataSim < SimulatorPkg.Simulator
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se

    properties (Access = private)
        structure % structure or FE model to be updated
        % activeDofs
        sparseMtx
        nrOfModes
        modalData
        mult
    end

    methods (Access = public)

        function obj = MatlabModalDataSim(...
            structure,...
            P0,...
            nrOfModes,...
            sparseMtx,...
            DEBUG)

            obj.structure = structure;
            obj.activeDofs = activeDofs;

            % clear old files
            clear(obj);

            obj.sparseMtx = sparseMtx;
            obj.nrOfModes = nrOfModes;

            update(obj,P0,SimulatorPkg.ResultType.GROUP_STIFF_mtx);
            run(obj,SimulatorPkg.ResultType.GROUP_STIFF_mtx);
        end
```
function update(obj, P, type)
    if type==SimulatorPkg.ResultType.MODAL_DATA || ...
        type==SimulatorPkg.ResultType.GROUP_STIFF_MTX
            update(obj, structure, P);
    end
end

function run(obj, type)
    % Run job in Abaqus
    if type==SimulatorPkg.ResultType.MODAL_DATA || ...
        type==SimulatorPkg.ResultType.GROUP_STIFF_MTX
            Kglob = getStiffMtx(obj, structure);
            Mglob = getMassMtx(obj, structure);
            % Kglob = Kglob(obj.activeDofs, obj.activeDofs);
            % Mglob = Mglob(obj.activeDofs, obj.activeDofs);
            if obj.sparseMtx
                Kglob = full(Kglob);
                Mglob = full(Mglob);
            end
    [V, D] = eig(Kglob, Mglob);
    lambda = diag(D);

    % NEW STAFF
    [aDofs, cDofs] = getActiveConstrainedDofs(obj, structure);
    obj.mult = []; ii = 1; m = 1;
    while m<=obj.nrOfModes
        A = Kglob - lambda(m)*Mglob;
        obj.mult(ii) = length(aDofs) - rank(A);
        m = m + obj.mult(ii);
        ii = ii + 1;
    end
    totalMult = sum(obj.mult);
    if totalMult>obj.nrOfModes
        nrOfModesUpd = totalMult;
    else
        nrOfModesUpd = obj.nrOfModes;
    end
    lambda = lambda(1:nrOfModesUpd);
    f = sqrt(lambda)./(2*pi);
    phi = zeros(length(aDofs)+length(cDofs), nrOfModesUpd);
    phi(aDofs,:) = V(:,1:nrOfModesUpd);
    obj.modalData = ModalData(f, lambda, phi, zeros(nrOfModesUpd,1), 1:nrOfModesUpd);
    % END NEW STAFF
end

function Result = getResult(obj, type)
    import AbaqusPkg.*
    if type==SimulatorPkg.ResultType.MODAL_DATA
        Result = obj.modalData;
    elseif type==SimulatorPkg.ResultType.GROUP_STIFF_MTX
        Result = getGrStiffMtx(obj, structure);
        if obj.sparseMtx
            % nrOfGroups = size(Result,1);
            % for ii=1:nrOfGroups
            %     Result{ii} = Result{ii}(obj.activeDofs, obj.activeDofs);
            end
        else
            Result = Result(obj.activeDofs, obj.activeDofs, :);
        end
    elseif type==SimulatorPkg.ResultType.GLOBAL_NODE_COORDS
        Result = getNodeCoords(obj, structure);
    elseif type==SimulatorPkg.ResultType.GROUP_ELEM_NODE_NRS
        Result = getElemNodeNrs(obj, structure);
    elseif type==SimulatorPkg.ResultType.GROUP_CENTER_COORDS
        Result = getGrCenterCoords(obj, structure);
    elseif type==SimulatorPkg.ResultType.GLOBAL_MASS
        Result = getMassMtx(obj, structure);
    elseif type==SimulatorPkg.ResultType.GLOBAL_STIF
        Result = getStiffMtx(obj, structure);
J.4.2 The MZCPlate class

Source code

```matlab
classdef MZCPlate < MatlabSimPkg.Structure
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
% MZC (Melosh, Zienkiewicz, Cheng)
% Plate elements for Kirchhoff plate model.
% Each element has 4 nodes with 3 dofs per node:
% 1. displacement along the thickness (z)
% 2 and 3 - rotations about x and y
% so totally 12 dofs per element
properties (Access = private)
    Kglob
    Mglob
    Ke0 % numerical element stiffness matrix without respect to
    % the corresponding elasticity modulus
    Me % numerical element mass matrix
    elem2GrNrMap
    elem2NodeMap
    node2CoordsMap
    nDofsPerNode
    nDofs
    nRowElems
    nColElems
    activeDofs
    constrainedDofs
    sparseMtx
methods
    function obj = MZCPlate(nRowElems, nColElems, a, b, Egr, nu, rho, t,
        elem2GrNrMap, sparseMtx)
% It is assumed that all elements are rectangular with
% dimensions (2a)x(2b), where a is the half of the length along
% x-axis and b is the half of the length along y-axis
% Deflections are continuous on the edges between elements but
% not the slopes of the deflections.
% INPUT:
% Plate is assumed to be rectangular
% nRowElems - number of elements in a row
% nColElems - number of elements in a column
% E - vector with elasticity modulus for each group
% nu - Poisson ratio
% rho - density
% t - thickness
% a - half of the x side of the element (all elements are assumed to be of equal size)
% b - half of the y side of the element (all elements are assumed to be of equal size)
% elem2GrNrMap - mapping between element number and group
% nRowElems - number of rows of elements
% nColElems - number of columns of elements
% nDofsPerNode - number of dofs per node
% nDofs - total number of dofs
% nRowElems - number of rows of elements
% nColElems - number of columns of elements
% activeDofs - dofs that are active
% constrainedDofs - dofs that are constrained
% sparseMtx - variable indicating if the composed
% matrices should be sparse or not. True - sparse, false - not
% OUTPUT:
% Kglob - global stiffness matrix
```

```matlab
function clear(obj)
    % TODO: warning ('Remove CREATED objects!'); -> logger
end
end
end
```
% Kglob - global stiffness matrix
% K0gr - group stiffness matrices
% Usage: [Kglob,Mglob] = assemblyKirchhoffPlate(20,20,200*10^9*ones(20*20,1)→
        ,0.2,7800,0.05,0.025,0.025);

% Elements are numbered from lower left corner to the right and then to the
% right upper corner, row by row.

% General settings
obj.sparseMtx = sparseMtx;
obj.elem2GrNrMap = elem2GrNrMap;
obj.nDofsPerNode = 3;
obj.nRowElems = nRowElems;
obj.nColElems = nColElems;

Mglob = getElemStiffMtx2(obj);
Ke = getElemStiffMtx2(obj);
Me = getElemStiffMtx3(obj);

% Find the sides of the element rectangle
%nodes = elem2NodeMap{1};
%[~,ind] = ismember(nodes,node2Coords(:,1));
%xy = node2Coords(ind,2:3);
[ke,me] = getElemMts(obj);
%ke = getElemStiffMtx1(obj);
%ke = getElemStiffMtx2(obj);
%ke = getElemStiffMtx3(obj);

% TEST
E = 200*10^9; nu=0.2; t=0.01;a=0.025;b=0.025;
Ke1 = double(subs(E*ke1));
Ke2 = double(subs(E*ke2));
Ke3 = double(subs(E*ke3));

% K0gr = cell(nGroups,1);
for gg=1:nGroups
  obj.K0gr{gg} = sparse(obj.nDofs,obj.nDofs);
end

% Find nodes which are the corners of the current element
nodes = obj.elem2NodeMap{e}; % node numbers
nGroup = elem2GrNrMap(e);
E = Egr(nGroup);
base = (nodes-1)*obj.nDofsPerNode; % offset corresponding to nodes
dofs = [base+1 base+2 base+3]; % dofs corresponding to nodes
dofs = dofs';
Ke0 = double(subs(Ke0=0));

if sparseMtx
  obj.K0gr{nGroup}(dofs,dofs) = obj.K0gr{nGroup}(dofs,dofs)+E*obj.Ke0;
else
  obj.K0gr(dofs,dofs,nGroup) = obj.K0gr(dofs,dofs,nGroup)+E*obj.Ke0;
end
function [elem2NodeMap, node2CoordsMap] = getMesh(obj, a, b)
nelem = obj.nRowElems*obj.nColElems;
 elem2NodeMap = cell(nelem,1);
node = (obj.nRowElems+1)∗(obj.nColElems+1);

node2CoordsMap = zeros(nnDoF,4);

end

for i=1:obj.nRowElems+1
  for j=1:obj.nColElems+1
    n1 = (i-1)+(obj.nRowElems+1)+j;
    n2 = n1 + 1;
    n3 = n4 + 1;
    elemNr = (i-1)*obj.nRowElems+j;
    elem2NodeMap(elemNr) = [n1;n2;n3;n4];
    nX = (j-1)*2+a;
    nY = (i-1)*2+b;
    nZ = 0;
    else
      nX = (j-1)*2+a;
      nY = (i-1)*2+b;
      nZ = 0;
      if j==obj.nColElems+1 && i<=obj.nRowElems
        nNr = n2;
      elseif i==obj.nRowElems+1 && j<=obj.nColElems
        nNr = obj.nRowElems*(obj.nColElems+1)+j;
      else
        nNr = nnDoF;
      end
      node2CoordsMap(k,:) = [nNr nX nY nZ];
      k = k + 1;
  end
end

function [bnodes, inodes] = getBoundaryInteriorNodes(obj)

node = (obj.nRowElems+1)∗(obj.nColElems+1);

bnodes = 1:obj.nColElems+1;
bnodes = [bnodes 2:obj.nColElems+2:obj.nColElems+1:nnDoF];

bnodes = [bnodes obj.nColElems+2:obj.nColElems+1:nnDoF−obj.nColElems+1];
bnodes = [bnodes nnDoF−obj.nColElems+1:nnDoF−1];

inodes = setdiff((1:nnDoF)',bnodes); % interior nodes

end

function setBoundaryConditions(obj, constrainedDofs)

obj.constrainedDofs = sort(constrainedDofs);

end

function [activeDofs, constrainedDofs] = getActiveConstrainedDofs(obj)

activeDofs = obj.constrainedDofs;
constrainedDofs = obj.constrainedDofs;

end

function [P,C,Q,T,D,detJ] = initialize(obj)

syms x y a b t nu xi eta

% Find necessary transformation matrix
T = [1/a 2 0 0; 0 1/b 2 0; 0 0 1/(a*b)];

% Define constitutive matrix

% Note! E is not included! since we want that groups differ in
% elasticity modulus

% constitutive matrix Ref.[1] Eq.(5.12) and Ref.[1] Eq.(5.15b)
D = t^3/(12*(1−nu^2))∗[1 nu 0
nu 1 0
0 0 1]
% Ref. [1] Eq. (5.33b)

C = [ subs(P, {x, y}, {-1, -1})
    subs(P_y, {x, y}, {-1, -1}) % node 1
    subs(-P_x, {x, y}, {1, -1})
    subs(P, {x, y}, {1, 1})
    subs(P_y, {x, y}, {1, 1}) % node 2
    subs(-P_x, {x, y}, {1, 1})
    subs(P, {x, y}, {-1, 1})
    subs(P_y, {x, y}, {-1, 1}) % node 3
    subs(-P_x, {x, y}, {-1, 1})
    subs(P, {x, y}, {1, 1}) ];

function value = getMassMtx(obj)
    value = obj.Mglob(obj.activeDofs, obj.activeDofs);
end

function value = getStiffMtx(obj)
    value = obj.Kglob(obj.activeDofs, obj.activeDofs);
end

function value = getGrStiffMtx(obj)
    if obj.sparceMtx
        nrOfGroups = max(obj.elem2GrNrMap(:,2));
        value = cell(nrOfGroups,1);
        for ii=1:nrOfGroups
            value{ii} = obj.K0gr{ii}(obj.activeDofs, obj.activeDofs);
        end
    else
        value = obj.K0gr(obj.activeDofs, obj.activeDofs,:);
    end
end

function update(obj,P)
    nGroups = length(P);
    if obj.sparceMtx
        obj.Mglob = sparse(obj.nDofs, obj.nDofs);
        obj.Kglob = sparse(obj.nDofs, obj.nDofs);
    end
    else
        obj.Kglob = zeros(obj.nDofs);
    end

Q = [ 0 0 0 0 0 0 6*x 2*y 0 0 6*x*y 0
      0 0 0 0 0 0 2*x 6*y 0 0 6*x*y
      0 0 0 0 0 0 4*x 4*y 0 6*x^2 6*y^2 ];

P = [ 1 x y x^2 x*y y^2 x^3 x^2*y x*y^2 y^3 x^3*y x*y^3 ];
P_y = 1/b*diff(P,y);
P_x = 1/a*diff(P,x);

% computing stiffness matrix

% |
% / dx_dxi dy_dxi \ / a 0 \ / dx_deta dy_dxi \ / 0 b /
%| %
detJ = a*b;
end

function value = getGrStiffMtx(obj)
    if obj.sparceMtx
        nrOfGroups = max(obj.elem2GrNrMap(:,2));
        value = cell(nrOfGroups,1);
        for ii=1:nrOfGroups
            value{ii} = obj.K0gr{ii}(obj.activeDofs, obj.activeDofs);
        end
    else
        value = obj.K0gr(obj.activeDofs, obj.activeDofs,:);
    end
end

function update(obj,P)
    nGroups = length(P);
    if obj.sparceMtx
        obj.Mglob = sparse(obj.nDofs, obj.nDofs);
        obj.Kglob = sparse(obj.nDofs, obj.nDofs);
    end
    else
        obj.Kglob = zeros(obj.nDofs);
    end

% obj.Mglob = zeros(obj.nDofs);
% arranging shape function in a 4x3 matrix (each row corresponds to node)
N = subs(N,
{            % rewriting shape functions in terms of xi and eta (natural coordinates)
    Q = subs(Q,{x,y},{xi,eta});
    invC = inv(C);
    ke = detJ*invC'*simplify(int(int(Qxe'*D*Qxe, xi, -1, 1), eta, -1, 1))*invC;
    N = P/C;
    % arranging shape function in a 4x3 matrix (each row corresponds to node)
    N = reshape(N,3,4).
end

% arranging shape function in a 4x3 matrix (each row corresponds to node)
Q = subs(Q,{x,y},{xi,eta});

% periodic boundary conditions
BB = cell(nen,1);
for i = 1:nen
    BB{i} = T*[ diff(N(i,1),xi,2)  diff(N(i,2),xi,2)  diff(N(i,3),xi,1) ];
end
\[ J.4 \text{ MatlabSimPkg package} \]
\[ \begin{align*}
\text{function} \quad & \text{ke} = \text{getElemStiffMtx3}(\text{obj}) \\
\text{import UtilityPkg.} & \\
\text{sym} & \text{s a b n u t E}
\end{align*} \]
\[ \% \text{stiffness matrix Ref.} [1] \text{ Eq.}(5.45) \]
\[ \text{ke} = \text{simplify(int(int(B.'*dB*detJ, xi, -1, 1), eta, -1, 1));} \]

\[ \begin{align*}
\text{k1} & = \begin{bmatrix} 6 & 0 & -6a & 0 & -6a & -6 & 0 & -6a & -3 & 0 & -3a & 3 & 0 & -3a \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 8a'2 & 6a & 0 & 4a''2 & 3a & 0 & 2a'2 & -3a & 0 & 4a'a2 & 2 \\
0 & 0 & 0 & 0 & 6a & 6a & 3 & 0 & 3a & -3 & 0 & 3a & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 8a'a'2 & 3a & 0 & 4a'a'2 & -3a & 0 & 2a'a2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 6a & -6 & 0 & 6a & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix} \\
\text{k2} & = \begin{bmatrix} 6 & 6b & 0 & 3 & 3b & 0 & -3 & 3b & 0 & -6 & 6b & 0 \\
0 & 8b'2 & 0 & 3b & 4b''2 & 0 & -3b & 2b'2 & 0 & -6b & 4b''2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 6 & b & 6b & 0 & -6 & 6b & 0 & -3 & 3b & 0 \\
0 & 0 & 0 & 0 & 8b'2 & 0 & -6b & 4b''2 & 0 & -3b & 2b'2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 6 & -6b & 0 & 3 & -3b & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & -6b & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix} \\
k2 & = a/(6b''3)*GeneralUtilities.Triang2SymmMtx(k2); \\
k3 & = \begin{bmatrix} 1 & b & -a & -1 & -b & 0 & 1 & 0 & 0 & -1 & 0 & a \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix} \\
k3 & = nu/(2a*b)*GeneralUtilities.Triang2SymmMtx(k3); \\
k4 & = \begin{bmatrix} 2 & 1 & 3b & -3a & -21 & 3b & 3a & -21 & 3b & -3a & 3a & -21 & 3b \\
0 & 8b''2 & 0 & -3b & -8b''2 & 0 & 3b & 2b'2 & 0 & -3b & -2b'2 & 0 \\
0 & 0 & 8a'a'2 & 3a & 0 & -2a''2 & -3a & 0 & 2a'2 & 3a & 0 & -8a'a'2 \\
0 & 0 & 0 & 21 & 3b & 3a & -21 & 3b & -3a & 21 & -3b & -3a \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 8b''2 & 0 & -3b & -8b''2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 8a'a'2 & -3a & 0 & -8a'a'2 & 3a & 0 & 2a'a2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix} \\
\]
The Structure class

Source code

```matlab
% Interface for finite element model built in Matlab.
%
classdef Structure < handle
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
methods (Abstract)
    % Gets the group stiffness matrices.
    getGrStiffMtx(obj);
    % Gets global mass matrix.
    getMassMtx(obj);
    % Gets global stiffness matrix.
    getStiffMtx(obj);
    % Updates system matrices with respect to the array P
    % of updating parameters numbered according to group numbers.
    update(obj,P);
end
```
J.5 ModalDataPkg package

J.5.1 The AverageWeighting class

Source code

```matlab
classdef AverageWeighting < ModalDataPkg.WeightingMethod
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se

    properties
        weights
    end

    methods
        function obj = AverageWeighting(freqLen,shapesLen)
            nrOfMeaPnts = shapesLen/freqLen;
            obj.weights = [ ones(freqLen,1) ;
                            ones(shapesLen,1)./sqrt(nrOfMeaPnts) ];
        end
        function obj = setWeights(weights)
            obj.weights = weights;
        end
        function value = getWeights(obj)
            value = obj.weights;
        end
        function value=toString(obj)
            value = 'Average weighting of mode shapes and uniform weighting of eigen ←' ;
            'frequencies';
        end
    end
end
```

J.5.2 The DerRepeatedEigenvalues class

Source code

```matlab
classdef DerRepeatedEigenvalues < ModalDataPkg.GradientMethod
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    % Class for computing derivatives
    % of modal data with respect to the updating parameters
    % (damage parameters)
    % Method implemented here supposes that the eigenvalue problem may
    % contain repeated eigenvalues, but does not contain repeated eigenvalue
    % derivatives. For the later case further improvement is required, see
    % Friswell "The derivatives of repeated eigenvalues and their
    % associated eigenvectors"
    properties (Access = private)
        K0gr
        sparseMtx
        Mass % global mass matrix, remains unchanged
        Ku % unchanged part of the global stiffness matrix
    end

    methods (Access = public)
        function obj = DerRepeatedEigenvalues(K0gr,Ku,Mass,sparseMtx)
            obj.K0gr = K0gr;
            obj.sparseMtx = sparseMtx;
            obj.Mass = Mass;
            obj.Ku = Ku; % test construction against reading from file!
        end

        % INPUT: p - vector of the updating parameters
        function gradient = evaluateGradient(obj,modalData,p,K,freeDofs)
            import ModalDataPkg.●

            nuFEM = getNu(modalData);
            phiFEM = getPhi(modalData);
            phiFEM = phiFEM(freeDofs,:);
    end
end
```
\[ \lambda = (2 \pi \nu_{\text{FEM}})^2; \] % eigenvalues in the increasing order

% % indices to unique values
% [\sim, \text{uniqueInd}] = unique(\nu_{\text{FEM}});
% % duplicate indices
% duplicateInd = setdiff(1:length(\nu_{\text{FEM}}), \text{uniqueInd});
% % duplicate values
% duplicateValue = \nu_{\text{FEM}}(\text{duplicateInd});

[\text{N, M}] = size(\phi_{\text{FEM}});

% N - length of mode shapes
% M - number of modes
% E - number of updating parameters = (groups of) elements
if \text{obj.sparseMtx}
    \text{E} = \text{length}(\text{obj}.K0gr);
else
    \text{E} = \text{length}(\text{obj}.K0gr(1,1,:));
end

\text{d}_{\omega_{\text{m}}}^{\text{d ae}} = \text{zeros}(\text{M}, \text{E});
\text{d}_{\phi_{\text{m}}}^{\text{d ae}} = \text{zeros}(\text{N, M}, \text{E});

% % Compose global stiffness matrix
% % COMMENT: probably it is better to read this matrix from a file?
% \text{K} = \text{obj}.Ku;
% \text{for} \text{e}=1:\text{E} \% for each updating parameter
% \text{if} \text{obj.sparseMtx}
% \text{K0grMat} = \text{obj}.K0gr\{\text{e}\};
% \text{else}
% \text{K0grMat} = \text{obj}.K0gr(:, :, \text{e});
% \text{end}
% \text{K} = \text{K} - \lambda(\text{e}) \times \text{K0grMat};
% \text{end}
% \text{end}
% \text{n} = \text{size}(\phi_{\text{FEM}}, 1);

% Find eigenvalue multiplicities
% % Solution when using eigs function
% \text{m} = 1;
% \text{mult} = [] ; \text{ii} = 1;
% \text{while} \text{m} < \text{M}
% \% disp(['\text{Group: ','num2str(e)},'\text{ mode: ','num2str(m)\}'])
% \text{A} = \text{K} - \lambda(m) \times \text{obj}.Mass; \% K, M - global stiffness and mass matrices
% \text{if} \text{m} == \text{M} \%|| \text{abs}(\lambda(m+1)-\lambda(m)) > 0.00001
% \text{mult}(\text{ii}) = 1;
% \% else
% \text{if} \text{obj.sparseMtx}
% \text{Afull} = \text{full}(\text{A});
% \text{mult}(\text{ii}) = \text{N} \text{ - rank} (\text{Afull});
% \% else
% \text{mult}(\text{ii}) = \text{N} \text{ - rank} (\text{A});
% \% end
% \text{end}
% \text{m} = \text{m} + \text{mult}(\text{ii});
% \text{ii} = \text{ii} + 1;
% \% end

% Solution when using Abaqus eigenvalue solver
% % Note! eigenvalues are sorted in increasing order.
% % indices to unique values
% [\sim, \text{uniqueInd}] = unique(\nu_{\text{FEM}});
% % duplicate indices
% duplicateInd = setdiff(1:length(\nu_{\text{FEM}}), \text{uniqueInd});
% % duplicate values
% duplicateValue = \nu_{\text{FEM}}(\text{duplicateInd});

\text{m} = 1;
\text{mult} = [] ; \text{ii} = 1;
\text{len} = \text{length}(\text{uniqueInd});
\text{while} \text{m} < \text{M}
if m==M || (abs(lambda(m+1)-lambda(m)) > 0.00001)
    mult(ii) = 1;
else if ii==len
    mult(ii) = M-uniqueInd(len);
else
    mult(ii) = uniqueInd(ii+1)-uniqueInd(ii);
end
m = m + mult(ii);
ii = ii + 1;
end

% if obj.sparseMtx
Kfull = full(K);
Mfull = full(obj.Mass);
[PhiFEM,D] = eig(Kfull,Mfull);
lambdaNew = diag(D);
end

for e=1:E % for each updating parameter
    if obj.sparseMtx
        K0grMat=full(obj.K0gr{e});
    else
        K0grMat=obj.K0gr(:,e);
    end

    Ident = eye(N);
    m = 1; ii = 1;
    Kfull = full(K);
    Mfull = full(obj.Mass);
    [Phifull,Dfull] = eig(Kfull,Mfull);
    lambdaFull = diag(Dfull);
    while m<M
        ind = m:m+mult(ii)-1;
        %Phi = phiFEM(:,ind);
        Phi = Phifull(:,ind);
        [H,lambdaH] = eig(-Phi'*K0grMat*Phi);
        % lambdaH comes in increasing order
        d_omegam2_d_ae(ind,e) = diag(lambdaH); %
        % Compute eigenvector derivatives
        disp([’Group: ’,num2str(e),’ mode: ’,num2str(m)]);
        %A = K - lambda(m)*obj.Mass; % K, M - global stiffness and mass matrices
        Psi = Phi*H;
        A = Kfull-lambdaFull(m)*Mfull;
        %A = Kfull - lambdaNew(m)*Mfull; % K, M - global stiffness and mass matrices

    end

m1 = length(ind); % multiplicity of the current eigenvalue

% Indices for zero elements of v
indZeros = zeros(m1,1);
tmpPsi = Psi;
for i=1:m1
    [~,mi] = max(abs(tmpPsi(:,i))); % CHECK if abs is needed???
    tmpPsi(mi(1),:) = 0;
    indZeros(i) = mi(1);
end
%indNonZeros = setdiff(1:size(Psi,1),indZeros);
G = A;
G(:,indZeros) = Ident(:,indZeros);
G(indZeros,:) = Ident(indZeros,:);
V = zeros(N,m1);
for i=1:m1
% Compute vectors \( C \) composed into matrix \( C \)
% Compute off-diagonal elements of \( C \)
% Define \( \text{diff} \)

\[ C = \text{zeros}(m1); \]

\begin{verbatim}
for i=1:m1
  for j=1:m1
    if i ~= j
      d = lambdaH(j,j) - lambdaH(i,i);
      % C(j,i) = Psi(:,j)'*(K0grMat+lambdaH(i,i)*obj.Mass)*V(:,i)/d;
      C(j,i) = Psi(:,j)'*(K0grMat+lambdaH(i,i)*Mfull)*V(:,i)/d;
    else
      % Compute and assign diagonal elements of \( C \)
      % C(i,i) = -V(:,i)'*obj.Mass*Psi(:,i);
      C(i,i) = -V(:,i)'*Mfull*Psi(:,i);
    end
  end
end
\end{verbatim}

% Assign the derivatives for the eigenvectors associated with repeated eigenvalues
% d_phim_d_ae(:,ind,e) = V + Psi*C;

m = m+mult(ii); ii = ii+1;

\begin{verbatim}
% if ismember(muFEM(m),duplicateValue)
%  % Case of repeated eigenvalue
%  % duplicateValue = setdiff(duplicateValue, muFEM(m))
% ;
%  ind = find(muFEM==muFEM(m));
%  Phi = phiFEM(:,ind);
%  [H,lambdaH] = eig(-Phi'*K0grMat*Phi);
%  % lambdaH comes in increasing order
%  % d_omegam2_d_ae(ind,e) = diag(-H'*Phi'*K0grMat*Phi*H);
%  % d_omegam2_d_ae(ind,e) = diag(-H'*Phi'*K0grMat*Phi*H);
%  % H;
%  % lhs=sort(lambdaH); % check efficiency
%  % if min(abs(lhs(2:end)-lhs(1:end-1)))<1000*eps
%  %   error('The case with two coinciding is not yet implemented.')
%  % end
%  %
%  % Compute eigenvector derivatives
%  % Psi = Phi*H;
%  % % Compute vectors \( V \) composed into matrix \( V \)
%  % ml = length(ind);
%  % V = zeros(N,ml);
%  % % Indices for zero elements of \( V \)
%  % indZeros = zeros(ml,1);
%  % % tmpPsi = Psi;
%  % for i=1:ml
%  %   [~,mi] = max(abs(tmpPsi(:,i))); % CHECK if abs is needed ???
%  %   %while ismember(mi,indZeros)
%  %     tmpPsi(mi(1), :) = -Inf;
%  %   % [~,mi] = max(Psi(:,i));
%  % end
%  % indZeros(i) = mi(1);
%  % end
%  %indNonZeros = setdiff(1:size(Psi,1),indZeros);
%  % for i=1:ml
%  %   b = (K0grMat + lambdaH(i,i)*obj.Mass)*Psi(:,i);
%  % end
%  % G = A;
\end{verbatim}
```matlab
245  G(:, indZeros) = Ident(:, indZeros);
246  G(indZeros, :) = Ident(indZeros, :);
247  V = [];
248  V(:, i) = G;b;
249  end
250  % Compute vectors c composed into matrix C
251  % Compute off-diagonal elements of C
252  % Define diff
253  C = zeros(ml);
254  for i=1:ml
255      for j=1:ml
256          if i==j
257              d = lambdaH(j,j)-lambdaH(i,i);
258              C(j,i) = Psi(:,j)’*(K0grMat+lambdaH(i,i)←
259                  )*obj.Mass)*V(:,i)/d;
259          end
261  end
262  % Compute and assign diagonal elements of C
263  C(1:ml+1:ml^2) = diag(-V'*obj.Mass*Psi);
264  % Assign the derivatives for the eigenvectors
265  % associated with repeated eigenvalues
266  d_phim_d_ae(:, ind , e) = V + Psi*C;
267  elseif ismember(m, uniqueInd)
269  % Case of unique eigenvalue
271  d_omega m2_d_ae(m, e) = -phiFEM(:,m).*K0grMat*phiFEM←
272                  (:,m);
274  % Use Nelson method to find the derivatives of
275  % the eigenvectors associated to the unique
277  % eigenvalues
278  [~, mi] = max(abs(phiFEM(:,m)));
279  G(2:mi(1)) = [];
280  G(mi(1), :) = [];
281  f = (K0grMat+d_omega m2_d_ae(m, e)*obj.Mass)*phiFEM←
282                  (:,m);
283  f(mi(1)) = [];
285  V_Tilde = G\f;
286  V(1:mi(1)-1:mi(1)+1:N) = V_Tilde;
287  V = V(:,);
288  C = -V'*obj.Mass*phiFEM(:,m);
289  d_phim_d_ae(:, m, e) = V + C*phiFEM(:,m);
290  end
291  disp([’Group: ’, num2str(e)])
292 end
294  gradient = ModalDataPkg.GradientData(d_omega m2_d_ae, d_phim_d_ae);
296  end
298  function value=toString(obj)
299    value=’DerRepeatedEigenvalues’;
300 end
303 end
```

### J.5.3 The DiffUniformWeighting class

**Source code**

```matlab
classdef DiffUniformWeighting < ModalDataPkg.WeightingMethod
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
```
J.5.4 The FoxKapoor class

This class contains methods for computing the Fox-Kapoor Formulae (1.17) (page 102).

Source code

```matlab
classdef FoxKapoor < ModalDataPkg.GradientMethod
    properties
        K0gr
        sparseMtx
    end
    methods
        function obj = FoxKapoor(K0gr, sparseMtx)
            obj.K0gr = K0gr;
            obj.sparseMtx = sparseMtx;
        end
        function gradient = evaluateGradient(obj, modalData, params.K, matrixFreeDofs)
            nuFEM = getFreq(modalData);
            phiFEM = getPhi(modalData);
            if isempty(matrixFreeDofs)
                phiFEM = phiFEM(matrixFreeDofs,:);
            end
            [N,M]=size(phiFEM);
            if obj.sparseMtx
                E=length(obj.K0gr); % Number of (groups of) elements
            else
                E=length(obj.K0gr(1,1,:)); % Number of (groups of) elements
            end
            d_omega2_d_ae = zeros(M,E); % Temporary D-dependence.
            d_phim_d_ae = zeros(N,M,E);
            for e=1:E
                if e==1
                    K0grMat=obj.K0gr(e);
                else
                    K0grMat=obj.K0gr(:,e);
                end
            end
        end
end
```
end

% Compute (13) in "Damage assessment ..." without for loop:

\[
\Lambda_m = \text{diag}(\Phi_m^T \mathbf{K}_{0gr} \Phi_m)
\]

\[
\cdot (2\pi \nu_{FEM})^2 - (2\pi \nu_{FEM}(m))^2.
\]

% % % % % R E M O V E THIS % % % % % % %

% tmp = isnan(Lambda_mDiagonal);
% tmp2 = find(tmp>0);
% if ~isempty(tmp2)
% Lambda_mDiagonal(tmp) = zeros(length(tmp2),1);
% end
Lambda_mDiagonal(m) = 0;
Lambda_m = diag(Lambda_mDiagonal);

% sum of column vectors

\[
d_{\Phi_m\Phi_m}(\cdot, m, e) = \sum(\Phi_m \Lambda_m^\prime); 
\]

% if ~isempty(find(Lambda_mDiagonal==Inf))
% disp('here')
% end

end

% % Quick and dirty handling of potentially different Fox−Kapoor
% % right−hand sides for different d:
% % ind=find(abs(d_{omegam2}_d_{ae}(m, e, \cdot)) > 1000*eps);
% % v=abs(d_{omegam2}_d_{ae}(m, e, ind));
% % if (max(v)−min(v)) > 1000*eps)
% % error('Mode frequencies not correctly computed ...
% % ' in the multivariate case')
% else
% d_{omegam2}_d_{ae}(m, e, \cdot) = mean(d_{omegam2}_d_{ae}(m, e, ind));
% end
%warning('Check what to do with rigid body modes!!!')
%d_{\Phi_m\Phi_m}(\cdot, 1:6, \cdot) = zeros(N,6,E);
%d_{omegam2}_d_{ae}(1:6, \cdot) = zeros(6,E);

\text{gradient} = \text{ModalDataPkg.GradientData}(d_{omegam2}_d_{ae}, d_{\Phi_m\Phi_m});
% Quick & dirty removal of potentially different Fox−Kapoor
% frequency right−hand sides for different d:
%d_{omegam2}_d_{ae} = d_{omegam2}_d_{ae}(\cdot, 1);

function value = toString(obj)
value = 'Fox−Kapoor';
end
end

J.5.5 The FreqResidual class

Source code

\texttt{classdef FreqResidual < ModalDataPkg.ResidMethod}
% Copyright Niklas Grip and Natalia Sabourova \{Niklas.Grip,Natalia.Sabourova\}@ltu.se

\texttt{properties (Access = private) }
\texttt{nrOfGroups}
\texttt{sparseMtx \% SEEMS TO BE NOT USED HERE!!}
\texttt{\%nRef }
\texttt{end}

\texttt{methods }

\texttt{function obj = FreqResidual(nrOfGroups, sparseMtx) }
\texttt{obj.nrOfGroups = nrOfGroups; \% Number of (groups of) elements }
\texttt{obj.sparseMtx = sparseMtx; }
\texttt{end}

\texttt{function [r_f, r_s] = computeResid(obj, modalDataTest, modalDataFem) }
\texttt{nuMea = getFreq(modalDataTest); }
nuSim = getFreq(modalDataFem);

r_f = (nuSim.^2 - nuMea.^2)./nuMea.^2;

end

function value = evaluateJacobian(obj, modalDataTest, modalDataFem, gradientData)

value = evaluateJrUpper(obj, modalDataTest, gradientData);

function value = toString(obj)

value = 'Frequency residual';

end

J.5.6 The GenNelsonRepeatedEigendata class

Source code

classdef GenNelsonRepeatedEigendata < ModalDataPkg.GradientMethod

% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
% Class for computing derivatives of modal data with respect to
% the updating parameters (damage parameters)
% Method implemented here assumes that the eigenvalue problem may
% contain repeated eigenvalues, but does not contain repeated eigenvalue
% derivatives. For the later case further improvement is required, see
% Friswell "The derivatives of repeated eigenvalues and their
% associated eigenvectors".

properties (Access = private)

K0gr % group stiffness matrices, remain unchanged
Mass % global mass matrix, remains unchanged
sparseMtx % indicator if the matrices are sparse or not
E % number of updating group = parameters
end

dections (Access = public)

function obj = GenNelsonRepeatedEigendata(K0gr, Mass, sparseMtx)

obj.K0gr = K0gr;
obj.Mass = Mass;
obj.spareMtx = sparseMtx;
if obj.spareMtx
obj.E = length(obj.K0gr);
else
obj.E = length(obj.K0gr(1,1,:));
end
end

% INPUT:
% modalData - modal data, containing eigenvalues and eigenvectors.
% Note, that eigenvectors contained even the constrained dofs,
% which is necessary because the indices corresponding to the
% residuals are defined with respect to the full eigenvectors.
% K - current global stiffness matrix
% activeDofs - dofs corresponding to active degrees of freedom

dunction gradient = evaluateGradient(obj, modalData, K, mult, activeDofs)

import ModalDataPkg.*

PhiFEM = getPhi(modalData);
[N,M]=size(PhiFEM); % active and constrained dofs

PhiFEM = PhiFEM(activeDofs,:); % corresponding to active and constrained dofs

lambda = getLambda(modalData);

% N - length of mode shapes
% M – number of modes
% E – number of updating parameters = (groups of) elements

\[ d_{\text{omegan2}}_{\text{d ae}} = \text{zeros}(M, \text{obj.E}); \]
\[ d_{\text{phim}}_{\text{d ae active}} = \text{zeros}(N_{\text{active}}, M, \text{obj.E}); \] % derivatives of
% eigenvectors corresponding to active dofs only!

Ident = \text{eye}(N_{\text{active}}); % define identity matrix

\% Find eigenvalue multiplicities
\% 
m = 1;
\% mult = [ ]; ii = 1;
\% while mc<M
\% A = K – lambda(m)*\text{obj.Mass};
\% if \text{obj.sparseMtx}
\% mult(ii) = Nactive – \text{rank(full(A))};
\% else
\% mult(ii) = Nactive – \text{rank(A)};
\% end
\% m = m + mult(ii);
\% ii = ii + 1;
\% end
\% if m-mult(ii-1)>M % last eigenvalue is not of multiplicity 1
\% not enough eigenvalue data
\% M = m-mult(ii-1);
\% mult = mult(1:end-1);
\% end

\% for e=1:obj.E % for each updating parameter
\% 
\% if \text{obj.sparseMtx}
\% K0grMat=\text{obj.K0gr}\{e\};
\% else
\% K0grMat=\text{obj.K0gr}\{,:,:,e\};
\% end

\% Compute eigenvector derivatives
\% \Psi = \Phi*H;
\% A = K–lambda(m)*\text{obj.Mass};

% Compute vectors \(v\) composed into matrix \(V\)
\% m1 = \text{length}(\text{ind}); % multiplicity of the current eigenvalue
\% 
\% Indices for zero elements of \(v\)
\% indZeros = \text{zeros}(m1,1);
\% tmpPsi = \Psi;
\% for i=1:m1
\% [","m1] = \text{max(abs(tmpPsi(:,i)))};
\% tmpPsi(mi(1),:) = 0;
\% indZeros(mi) = mi(1);
\% end
%indNonZeros = \text{setdiff}(1:size(\Psi,1),indZeros);

G = A;
G(:,indZeros) = Ident(:,indZeros);
G(indZeros,:) = Ident(indZeros,:);

V = \text{zeros}(N_{\text{active}},m1);
for i=1:m1
f = (K0grMat + lambdaH(i,i)*\text{obj.Mass})*\Phi(:,i);
f(indZeros) = \text{zeros}(m1,1);
V(:,i) = G\{f;
end

% Compute vectors \(c\) composed into matrix \(C\)
% Compute off–diagonal elements of \(C\)
176
\textbf{J.5.7 The GenNelsonRepeatedEigenvalues class}

\textbf{Source code}

```matlab
classdef GenNelsonRepeatedEigenvalues < ModalDataPkg.GradientMethod
\textcopyright Niklas Grip and Natalia Sabourova \{Niklas.Grip,Natalia.Sabourova\}@ltu.se
% Class for computing derivatives
% of modal data with respect to the updating parameters
% Method implemented here supposes that the eigenvalue problem may
% contain repeated eigenvalues, but does not contain repeated eigenvector
derivatives. For the later case further improvement is required, see
% Friswell "The derivatives of repeated eigenvalues and their
% associated eigenvectors".
properties (Access = private)
    K0gr
    sparseMtx
    Mass \% global mass matrix, remains unchanged
    Ku \% unchanged part of the global stiffness matrix
end
methods (Access = public)
function obj = GenNelsonRepeatedEigenvalues(K0gr,Ku,Mass,sparseMtx)
    obj.K0gr = K0gr;
    obj.sparseMtx = sparseMtx;
    obj.Mass = Mass;
    obj.Ku = Ku; \% test construction against reading from file!
end
% INPUT: p - vector of the updating parameters
function gradient = evaluateGradient(obj,modalData,p,K.freeDofs)
import ModalDataPkg.*
    nuFEM = getFreq(modalData);
    PhiFEM = getPhi(modalData);
    phiFEM = PhiFEM(freeDofs,:);
    lambda = getLambda(modalData); \% unfortunately this approximation
    \% is not enough for the construction of matrix A below
```
if obj.sparseMtx
    %[N,M]=size(PhiFEM);
    E=length(obj.K0gr);
    [~,D]=eigs(K,obj.Mass,M,'sm');
    %[PhiFEM,D]=eigs(K,obj.Mass,M,'sm');% it is working without this
    lambda=diag(D);
    [~,D]=sort(lambda);% in increasing order
    %PhiFEM = PhiFEM(:,ind);
else
    E=length(obj.K0gr(1,1,:));
    [~,D]=eig(K,obj.Mass);
    %[PhiFEM,D]=eig(K,obj.Mass);% it is working without this
    lambda=diag(D);
end

% N − length of mode shapes
% M − number of modes
% E − number of updating parameters = (groups of) elements

d_omega2_d_ae=zeros(M,E);
d_phim_d_ae=zeros(N,M,E);

% Find eigenvalue multiplicities
% Solution when using eigs function

m = 1;
mult = [];
ii = 1;
while m <= M
    %disp(’Group: ’,num2str(e),’ mode: ’,num2str(m))
    A = K - lambda(m)*obj.Mass; % K, M − global stiffness and mass matrices
    if obj.sparseMtx
        mult(ii) = N - rank(full(A));
    else
        mult(ii) = N - rank(A);
    end
    m = m + mult(ii);
    ii = ii + 1;
end

if m-1>M % not enough eigenvalue data
    M = m-mult(ii-1);
    mult = mult(1:end-1);
end

for e=1:E % for each updating parameter
    if obj.sparseMtx
        K0grMat=obj.K0gr{e};
    else
        K0grMat=obj.K0gr(:,e);
    end

    m = 1; ii = 1;
    while m<M
The GenNelsonRepeatedEigenvaluesNew class

Source code

```matlab
classdef GenNelsonRepeatedEigenvaluesNew < ModalDataPkg.GradientMethod
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    % Class for computing derivatives
    % of modal data with respect to the updating parameters
    % (damage parameters)
    % Method implemented here supposes that the eigenvalue problem may
    % contain repeated eigenvalues, but does not contain repeated eigenvalue
    % derivatives. For the latter case further improvement is required, see
    % Friswell "The derivatives of repeated eigenvalues and their
    % associated eigenvectors".
    properties (Access = private)
        K0gr
        sparseMtx
        Mass % global mass matrix, remains unchanged
        Ku % unchanged part of the global stiffness matrix
    end
    methods (Access = public)
        function obj = GenNelsonRepeatedEigenvaluesNew(K0gr,Ku,Mass,sparseMtx)
            obj.K0gr = K0gr;
            obj.sparseMtx = sparseMtx;
            obj.Mass = Mass;
            obj.Ku = Ku; % test construction against reading from file!
        end

        % INPUT: p − vector of the updating parameters
        function gradient = evaluateGradient(obj,modalData,K,activeDofs)
            import ModalDataPkg.*
            if obj.sparseMtx
                E=length(obj.K0gr);
            else
                E=length(obj.K0gr(1,1,:));
            end
            %nuFEM = getFreq(modalData);
            PhiFEM = getPhi(modalData); % expanded to contain constrained dofs!!!
            [N,M]=size(PhiFEM);
            d_phim_d_ae = zeros(N,M,E);
            PhiFEM = PhiFEM(activeDofs,:); % NEW
            lambda = getLambda(modalData);
            % is not enough for the construction of matrix A below
            [N,M]=size(PhiFEM);
```
ModalDataPkg package

% N – length of mode shapes
% M – number of modes
% E – number of updating parameters = (groups of) elements

d_omegaM2_d_au = zeros(M,E);
d_phim_d_au_active = zeros(N,M,E);

% Find eigenvalue multiplicities
% Solution when using eigs function
m = 1;
mult = [] ; ii = 1;
while m<M
    %disp(['Group: ', num2str(e), ' mode: ', num2str(m)])
    A = K - lambda(m)*obj.Mass; % K, M - global stiffness and mass matrices
    if obj.sparseMtx
        mult(ii) = N - rank(full(A));
    else
        mult(ii) = N - rank(A);
    end
    m = m + mult(ii);
    ii = ii + 1;
end
if m<M
    % not enough eigenvalue data
    M = m - mult(ii-1);
    mult = mult(1:end-1);
end
% Solution when using Abaqus eigenvalue solver
% Note! eigenvalues are sorted in increasing order.
% % indices to unique values
% [~, uniqueInd] = unique(mFEM);
% m = 1;
% mult = [] ; ii = 1;
len = length(uniqueInd);
while m<M
    if m>M %|| (abs(lambda(m+1)-lambda(m))>0.00001)
        mult(ii) = 1;
    elseif ii==len
        mult(ii) = M-uniqueInd(len);
    else
        mult(ii) = uniqueInd(ii+1)-uniqueInd(ii);
    end
    m = m + mult(ii);
    ii = ii + 1;
end

for e=1:E % for each updating parameter
    if obj.sparseMtx
        K0grMat = obj.K0gr{e};
    else
        K0grMat = obj.K0gr(:, :, e);
    end
    m = 1; ii = 1;
    while m<M
        ind = m + mult(ii) - 1;
        Phi = PhiFEM(:, :, ind);
        [~, lambdaH] = eig(-Phi'*K0grMat*Phi);
        % lambdaH comes in increasing order
        d_omegaM2_d_au(ind, e) = diag(lambdaH);
        m = m + mult(ii); ii = ii + 1;
    end
    %disp(['Group: ', num2str(e)])
end

d_phim_d_au(activeDofs, :, :) = d_phim_d_au_active;
gradient = ModalDataPkg.GradientData(d_omegaM2_d_au, d_phim_d_au);

J.5.9 The GenNelsonRepeatedEigenvaluesNew class

Source code

```matlab
function value=toString(obj)
    value='GenNelsonRepeatedEigenvalues';
end
end

function obj = GenNelsonRepeatedEigenvaluesNew(K0gr, Ku, Mass, sparseMtx)
    obj.K0gr = K0gr; obj.sparseMtx = sparseMtx; obj.Mass = Mass; obj.Ku = Ku; % test construction against reading from file!
end

function gradient = evaluateGradient(obj, modalData, K, activeDofs)
    import ModalDataPkg.*
    if obj.sparseMtx
        E=length(obj.K0gr);
    else
        E=length(obj.K0gr(1,1,:));
    end

    % nuFEM = getFreq(modalData);
    PhiFEM = getPhi(modalData); % expanded to contain constrained dofs!!!
    [N,M]=size(PhiFEM);
    d_phim_d_ae = zeros(N,M,E);
    PhiFEM = PhiFEM(activeDofs,:); % NEW

    lambda = getLambda(modalData);
    % is not enough for the construction of matrix A below
    [N,M]=size(PhiFEM);
    % N - length of mode shapes
    % M - number of modes
    % E - number of updating parameters = (groups of) elements
    d_omegam2_d_ae = zeros(M,E);
    d_phim_d_ae_active = zeros(N,M,E);

    % Find eigenvalue multiplicities
    % Solution when using eigs function
    m = 1;
    mult = []; ii = 1;
    while ii<=M
        %disp(['Group: ',num2str(e),' mode: ',num2str(m)])
```
J.5 ModalDataPkg package

Source code

```matlab
A = K - lambda(m)*obj.Mass; % K, M - global stiffness and mass matrices
if obj.sparseMtx
    mult(ii) = N - rank(full(A));
else
    mult(ii) = N - rank(A);
end
m = m + mult(ii);
i = i + 1;
end
if m>M
    % not enough eigenvalue data
    M = m-mult(ii)-1;
end
% Solution when using Abaqus eigenvalue solver
% % Note! eigenvalues are sorted in increasing order.
% % indices to unique values
% [~, uniqueInd] = unique(mFEM);

m = 1;
while m<M
    if m==M || (abs(lambda(m+1)-lambda(m))>0.00001)
        mult(ii) = 1;
    elseif i==len
        mult(ii) = M-uniqueInd(len);
    else
        mult(ii) = uniqueInd(ii+1)-uniqueInd(ii);
    end
    m = m + mult(ii);
i = i + 1;
end
% for e=1:E % for each updating parameter
if obj.sparseMtx
    K0grMat = obj.K0{e};
else
    K0grMat = obj.K0(:,:,:); 
end
m = 1; i = 1;
while m<=M
    ind = m:mult(ii)-1;
    Phi = PhiFEM(:,ind);
    [~, lambdaH] = eig(-Phi*K0grMat*Phi);
    % lambdaH comes in increasing order
    d_omegam2_d_ae(ind,e) = diag(lambdaH); %
    m = m+mult(ii);
i = i+1;
end
end
end
end
%disp(['Group: ',num2str(e)])
d_phim_d_ae(activeDofs,:,:,:) = d_phim_d_ae_active;
gradient = ModalDataPkg.GradientData(d_omegam2_d_ae,d_phim_d_ae);
end
function value=toString(obj)
value = 'GenNelsonRepeatedEigenvalues';
end
end
```

J.5.10 The GradientData class

Source code
J.5.11 The GradientMethod class

Source code

% The source code

classdef GradientMethod < handle

% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
% Abstract class for creating data graphs
% Subclass constructor should accept
% the data that is to be plotted and
% property name/property value pairs
methods (Abstract)
%evaluateGradient(obj,modalData,params,K mult,matrixFreeDofs)
evaluateGradient(obj,modalData,K mult,matrixFreeDofs)
toString(obj)
% Compute the gradient
end
% methods
% function set.Data(obj,newdata)
% obj.Data = newdata;
% updateGraph(obj)
end
% function addButtons(gobj)
% hfig = get(gobj.AxesHandle,'Parent');
% uicontrol(hfig,'Style','pushbutton','String','Zoom Out',....
% 'Callback',@(src,evnt)zoom(gobj,.5));
% uicontrol(hfig,'Style','pushbutton','String','Zoom In',....
% 'Callback',@(src,evnt)zoom(gobj,2),....
% 'Position',[100 20 60 20]);
% end
% end
end

J.5.12 The IdentityPairing class

Source code

% The source code

classdef IdentityPairing < handle

% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
% Abstract class for creating data graphs
% Subclass constructor should accept
% the data that is to be plotted and
% property name/property value pairs
methods (Abstract)
%evaluateGradient(obj,modalData,params,K mult,matrixFreeDofs)
evaluateGradient(obj,modalData,K mult,matrixFreeDofs)
toString(obj)
% Compute the gradient
end
% methods
% function set.Data(obj,newdata)
% obj.Data = newdata;
% updateGraph(obj)
end
% function addButtons(gobj)
% hfig = get(gobj.AxesHandle,'Parent');
% uicontrol(hfig,'Style','pushbutton','String','Zoom Out',....
% 'Callback',@(src,evnt)zoom(gobj,.5));
% uicontrol(hfig,'Style','pushbutton','String','Zoom In',....
% 'Callback',@(src,evnt)zoom(gobj,2),....
% 'Position',[100 20 60 20]);
% end
% end
end

% The source code

classdef GradientData < handle

% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
% The source code

properties (Access = private)
d_omegam2_d_ae
d_phim_d_ae
end
methods
function obj = GradientData(d_omegam2_d_ae,d_phim_d_ae)
obj.d_omegam2_d_ae = d_omegam2_d_ae;
obj.d_phim_d_ae = d_phim_d_ae;
end
function value = getD_omegam2_d_ae(obj)
value = obj.d_omegam2_d_ae;
end
function value = getD_phim_d_ae(obj)
value = obj.d_phim_d_ae;
end
function setD_omegam2_d_ae(obj,d_omegam2_d_ae)
obj.d_omegam2_d_ae = d_omegam2_d_ae;
end
function setD_phim_d_ae(obj,d_phim_d_ae)
obj.d_phim_d_ae = d_phim_d_ae;
end
function obj = getExtracted(obj,modeInd,shapeInd)
obj.d_omegam2_d_ae = obj.d_omegam2_d_ae(modeInd,:);
obj.d_phim_d_ae = obj.d_phim_d_ae(shapeInd,modeInd,:);
end
end
end
classdef IdentityPairing < ModalDataPkg.ModePairingMethod
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
methods (Access = public)
  function Mea2FEMind = pairModes(obj,modalDataFem,modalDataTest)
    % USAGE: function [BestMatchingFEMind,phiFEM,nuFEM,phiMea,nuMea] = 
    % ModePairingIndices(phiFEM,nuFEM,phiMea,nuMea)
    nTest = length(getFreq(modalDataTest));
    nFem = length(getFreq(modalDataFem));
    n = min(nTest,nFem);
    Mea2FEMind = [(1:n)' (1:n)'];
  end
  function value=toString(obj)
    value='Identity pairing method';
  end
end

J.5.13 The L2NormResidual class

Source code

classdef L2NormResidual < ModalDataPkg.ResidMethod
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
properties (Access = private)
  nrOfGroups
  sparseMtx % SEEMS TO BE NOT USED HERE!!
% nRef
end
methods
  function obj = L2NormResidual(nrOfGroups,sparseMtx)
    obj.nrOfGroups = nrOfGroups; % Number of ( groups of ) elements
    obj.sparseMtx = sparseMtx;
  end
  function [r_f r_s] = computeResid(obj,modalDataTest,modalDataFem)
    nuMea = getNu(modalDataTest);
    nuSim = getNu(modalDataFem);
    phiMea = getPhi(modalDataTest);
    phiSim = getPhi(modalDataFem);
    [N,M]=size(phiSim);
    r_f = (nuSim.'^2 - nuMea.^2)./nuMea.^2;
    r_s = zeros(N,M);
    for mm = 1:M
      normPHIfem = sqrt(phiFEM(:,:mm,dd)'*phiFEM(:,:mm,dd));
      normPHIoma = sqrt(phiOMA(:,:mm,dd)'*phiOMA(:,:mm,dd));
      normPHIsim = norm(phiSim(:,mm));
      normPHImea = norm(phiMea(:,mm));
      r_s(:,mm) = phiSim(:,mm)./normPHIsim - phiMea(:,mm)./normPHImea;
    end
    r_f = r_f(:); % Make column vector
    r_s = r_s(:); % Make column vector
    r = [r_f(:); r_s(:)]; % The residual
    disp('Freqs/Shapes')
    disp([sum(r_f.^2) sum(r_s.^2)])
  end
  function value = evaluateJacobian(obj,modalDataTest,modalDataFem,gradientData)
    JrUpper = evaluateJrUpper(obj,modalDataTest,gradientData);
    [N,M]=size(getPhi(modalDataFem));
    phiFEM = getPhi(modalDataFem);
    JrLower=zeros(N,M,obj.nrOfGroups);
The MACPairing class does pairing of modes using the public method `pairModes` and some additional private methods.

### Source code

```matlab
classdef MACPairing < ModalDataPkg.ModePairingMethod
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    methods (Access = public)
        function Mea2FEMind = pairModes(obj, modalDataFem, modalDataTest)
            % USAGE: function [BestMatchingFEMind, phiFEM, nuFEM, phiMea, nuMea] = ...
            % ModePairingIndices(phiFEM, nuFEM, phiMea, nuMea)
            % (Updated January 21 2014 by Niklas)
            % Note! It is required that the corresponding mode shapes have
            % the same length, otherwise the pairing fails. On the other
            % hand the number of mode shapes can be different in test and
            % fem modal data.

            % Inputs
            % phiFEM
            % phiMea
            % Outputs
            % Mea2FEMind = A two column matrix. If the elements on one row is [a,b],
            % then that indicates that the measured mode number a has
            % been paired with the predicted mode number b

            nuFEM = getFreq(modalDataFem);
            phiFEM = getPhi(modalDataFem);
            nuMea = getFreq(modalDataTest);
            phiMea = getPhi(modalDataTest);

            nFem = length(phiFEM);
            nTest = length(phiMea);

            if nFem ~= nTest
                error('Mode pairing fails: mode shapes have different lengths!')
            end
            % false
            nfe = obj.nrOfGroups
            for mm = 1:M
                normPHIfem = norm(phiFEM(:,mm));
                JrLower(:,mm) = d_phim_d_ae(:,mm)./normPHIfem ... 
                = (d_phim_d_ae(:,mm))/(normPHIfem*3) ... 
                *phiFEM(:,mm);
            end
            JrLower = reshape(JrLower, N*nM, obj.nrOfGroups);
            value = [JrUpper JrLower];
    end
end
end
```

J.5.14 The MACPairing class

The MACPairing class does pairing of modes using the public method `pairModes` and some additional private methods.
noPredModes = size(phiFEM,2); % Number of predicted modes from FE model

% MAC criteria for pairing of mode shapes
% Row index indicates the observed (measured) mode shape
% Column index indicates the predicted (FEM) mode shape
MACt = zeros(nofObsModes,nofPredModes); % Table of MAC values
for oo=1:nofObsModes
  for pp=1:nofPredModes
    MACt(oo,pp) = mac(obj.phiMea(:,oo),phiFEM(:,pp)); % Defined below
  end;
end;

Mea2FEMind = greedyPickLarge(obj,MACt);

% Mode pairing with respect to modeshapes done.
% The observed mode shapes (phiMEAS) are now sorted after increasing
% frequency (nuMea), but the same might not hold for the predicted
% mode shapes
% Code will follow here for warning messages when the predicted modes not
% comes in increasing frequency order.
% [dummy,ind] = sort(Mea2FEMind(:,1));
% [dummy,ind2] = sort(Mea2FEMind(ind,:));
% for ii = 1:length(ind2)-1
%   if ind2(ii) > ind(ii+1)
%     warning('xxx'
%   end
%
% end
%
% warning('Mode pairing currently checks mode shapes only');

function value=toString(obj)
value='MAC pairing method';
end

methods (Access = private)

% Modal assurance criterion for vectors u and v
function MACuv = mac(obj,u,v)
  MACuv = abs(u'*v)^2 / (u'*u * v'*v);
end

function indRC=greedyPickLarge(obj,A)
% Greedy algorithm for finding large elements in the matrix A without
% two of them being on the same row or column
% INPUTS
% A = rxc matrix
% OUTPUTS
% indRC = max(r,c)x2 matrix with each row containing the row- and column
% index of one chosen element
[nofR,nofC]=size(A);
indR=1:nofR;
indC=1:nofC;
nofElements2pick=min(nofR,nofC);
indRC=zeros(nofElements2pick,2); % Reserve memory

ElementCtr=1; % Counter for next element to pick
while (ElementCtr<=nofElements2pick)
  [maxA,r,c]=rcMax(obj,A);
  indRC(ElementCtr,:)=[indR(r),indC(c)];
  A = removeRowCol(obj,A,r,c);
  ]
end

function [dummy,ind] = sort(A)

function [dummy,ind] = rcMax(obj,A)

function value=toString(obj)
value='MAC pairing method';
end

methods (Access = private)

% Modal assurance criterion for vectors u and v
function MACuv = mac(obj,u,v)
  MACuv = abs(u'*v)^2 / (u'*u * v'*v);
end

function indRC=greedyPickLarge(obj,A)
% Greedy algorithm for finding large elements in the matrix A without
% two of them being on the same row or column
% INPUTS
% A = rxc matrix
% OUTPUTS
% indRC = max(r,c)x2 matrix with each row containing the row- and column
% index of one chosen element
[nofR,nofC]=size(A);
indR=1:nofR;
indC=1:nofC;
nofElements2pick=min(nofR,nofC);
indRC=zeros(nofElements2pick,2); % Reserve memory

ElementCtr=1; % Counter for next element to pick
while (ElementCtr<=nofElements2pick)
  [maxA,r,c]=rcMax(obj,A);
  indRC(ElementCtr,:)=[indR(r),indC(c)];
  A = removeRowCol(obj,A,r,c);
  ]
end

function [dummy,ind] = sort(A)

function [dummy,ind] = rcMax(obj,A)

function value=toString(obj)
value='MAC pairing method';
end

methods (Access = private)

% Modal assurance criterion for vectors u and v
function MACuv = mac(obj,u,v)
  MACuv = abs(u'*v)^2 / (u'*u * v'*v);
end

function indRC=greedyPickLarge(obj,A)
% Greedy algorithm for finding large elements in the matrix A without
% two of them being on the same row or column
% INPUTS
% A = rxc matrix
% OUTPUTS
% indRC = max(r,c)x2 matrix with each row containing the row- and column
% index of one chosen element
[nofR,nofC]=size(A);
indR=1:nofR;
indC=1:nofC;
nofElements2pick=min(nofR,nofC);
indRC=zeros(nofElements2pick,2); % Reserve memory

ElementCtr=1; % Counter for next element to pick
while (ElementCtr<=nofElements2pick)
  [maxA,r,c]=rcMax(obj,A);
  indRC(ElementCtr,:)=[indR(r),indC(c)];
  A = removeRowCol(obj,A,r,c);
  ]
end

function [dummy,ind] = sort(A)

function [dummy,ind] = rcMax(obj,A)

function value=toString(obj)
value='MAC pairing method';
end

methods (Access = private)

% Modal assurance criterion for vectors u and v
function MACuv = mac(obj,u,v)
  MACuv = abs(u'*v)^2 / (u'*u * v'*v);
end

function indRC=greedyPickLarge(obj,A)
% Greedy algorithm for finding large elements in the matrix A without
% two of them being on the same row or column
% INPUTS
% A = rxc matrix
% OUTPUTS
% indRC = max(r,c)x2 matrix with each row containing the row- and column
% index of one chosen element
[nofR,nofC]=size(A);
indR=1:nofR;
indC=1:nofC;
nofElements2pick=min(nofR,nofC);
indRC=zeros(nofElements2pick,2); % Reserve memory

ElementCtr=1; % Counter for next element to pick
while (ElementCtr<=nofElements2pick)
  [maxA,r,c]=rcMax(obj,A);
  indRC(ElementCtr,:)=[indR(r),indC(c)];
  A = removeRowCol(obj,A,r,c);
  ]
end

function [dummy,ind] = sort(A)

function [dummy,ind] = rcMax(obj,A)

function value=toString(obj)
value='MAC pairing method';
end

methods (Access = private)

% Modal assurance criterion for vectors u and v
function MACuv = mac(obj,u,v)
  MACuv = abs(u'*v)^2 / (u'*u * v'*v);
end

function indRC=greedyPickLarge(obj,A)
% Greedy algorithm for finding large elements in the matrix A without
% two of them being on the same row or column
% INPUTS
% A = rxc matrix
% OUTPUTS
% indRC = max(r,c)x2 matrix with each row containing the row- and column
% index of one chosen element
[nofR,nofC]=size(A);
indR=1:nofR;
indC=1:nofC;
nofElements2pick=min(nofR,nofC);
indRC=zeros(nofElements2pick,2); % Reserve memory

ElementCtr=1; % Counter for next element to pick
while (ElementCtr<=nofElements2pick)
  [maxA,r,c]=rcMax(obj,A);
  indRC(ElementCtr,:)=[indR(r),indC(c)];
  A = removeRowCol(obj,A,r,c);
  ]
end

function [dummy,ind] = sort(A)

function [dummy,ind] = rcMax(obj,A)
% Test code
% A=rand(7,7)
% indRC=greedyPickLarge(A)

function [maxA, r, c] = rcMax(obj, A);
% Find the largest element in the matrix A, as well as its row- and column
% index of
[rows, cols] = size(A);
if (rows==1)
  r = 1;
  [maxA, c] = max(A);
elseif (cols==1)
  c = 1;
  [maxA, r] = max(A);
else
  [colMax, RowInds] = max(A); % Max of each column and the row index for each
  [maxA, c] = max(colMax);
  r = RowInds(c);
end

% Test code
% A = rand(4, 4)
% [maxA, r, c] = rcMax(A)
% A = rand(1, 4)
% [maxA, r, c] = rcMax(A)
% A = rand(4, 1)
% [maxA, r, c] = rcMax(A)

function vR = removeElem(obj, v, n)
% Removes element number n from input vector/matrix v and returns the
% remaining elements in the column vector vR
v = v(:); % Make column vector
vLen = length(v);
if (1 <= n && n <= vLen)
vR = v([1:n-1 n+1:vLen]);
else
  error('Input arguments n not in the right range!')
end

function Ar = removeRowCol(obj, A, r, c)
% Removes row number r and column number c from the matrix A
[nofR, ofC] = size(A);
if (1 <= r && r <=nofR) && (1 <= c) && (c <= ofC)
  Ar = A([1:r-1 r+1:nofR], [1:c-1 c+1:ofC]);
else
  error('Input arguments r and c not in the right range!')
end
end
end
end

J.5.15 The MACResidual class

Source code

```matlab
classdef MACResidual < ModalDataPkg.ResidMethod
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip, Natalia.Sabourova}@ltu.se
```
properties (Access = private)
    nrOfGroups
    sparseMtx \% SEEMS TO BE NOT USED HERE!!
    \%nRef
end

methods

function obj = MACResidual(nrOfGroups, sparseMtx)
    obj.nrOfGroups = nrOfGroups; \% Number of ( groups of ) elements
    obj.sparseMtx = sparseMtx;
end

function [r_f, r_s] = computeResid(obj, modalDataTest, modalDataFem)
    nuMea = getNu(modalDataTest);
    nuSim = getNu(modalDataFem);
    phiMea = getPhi(modalDataTest);
    phiSim = getPhi(modalDataFem);
    r_f = (nuSim.^2 - nuMea.^2) ./ nuMea.^2;
    normPhiMea2 = sum(abs(phiMea).^2,1);
    normPhiSim2 = sum(abs(phiSim).^2,1);
    r_s = 1 - abs(dot(phiSim, phiMea,1)).^2 ./ (normPhiMea2.*normPhiSim2);
    r_f = r_f(:); \% Make column vector
    r_s = r_s(:); \% Make column vector
end

function value = evaluateJacobian(obj, modalDataTest, modalDataFem, gradientData)
    JrUpper = evaluateJrUpper(obj, modalDataTest, gradientData);
    phiSim = getPhi(modalDataFem);
    [\, M]=size(phiSim);
    phiMea = getPhi(modalDataTest);
    JrLower=zeros(M, obj.nrOfGroups);
    d_phim_d_ae = getD_phim_d_ae(gradientData);
    normPhiMea2 = sum(abs(phiMea).^2,1);
    normPhiSim2 = sum(abs(phiSim).^2,1);
    A = dot(phiMea, phiSim,1);
    for ee=1:obj.nrOfGroups
        B = dot(phiMea, d_phim_d_ae(:, :, ee),1);
        D = dot(phiSim, d_phim_d_ae(:, :, ee),1);
        %C = normPhiSim2(:,ee);
        JrLower(:, ee) = -2*A.*B.*C
        ./ (normPhiMea2.*normPhiSim2.^2);
    end
    JrLower=reshape(JrLower, M, obj.nrOfGroups);
    value = [JrUpper
              JrLower];
end

function value=toString(obj)
    value='MAC residual';
end
end

J.5.16 The ModalData class

Contains the methods ModalData, getNu, getPhi, getModeNrs, setNu, setPhi, setModeNrs
for setting and getting frequencies $\nu_m$ and mode shapes $\phi_{m,d}$ for an object of type ModalData.
Source code

classdef ModalData < handle
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se

properties (Access = private)
  damping
  f % eigenfrequencies
  phi % eigenvectors
  lambda % eigenvalues
  modeNrs
end

methods
  function obj = ModalData(f,lambda,phi,damping,modeNrs)
    if (length(f)~=size(phi,2) || length(f)~=length(damping) ||
        length(f)~=length(modeNrs))
      error('Inconsistent modal data');
    end
    obj.f = f;
    obj.lambda = lambda;
    obj.phi = phi;
    obj.damping = damping;
    obj.modeNrs = modeNrs;
  end
  function value = getFreq(obj)
    value = obj.f;
  end
  function value = getLambda(obj)
    value = obj.lambda;
  end
  function value = getPhi(obj)
    value = obj.phi;
  end
  function value = getDamping(obj)
    value = obj.damping;
  end
  function value = getModeNrs(obj)
    value = obj.modeNrs;
  end
  function setFreq(obj,f)
    obj.f = f;
  end
  function setLambda(obj,lambda)
    obj.lambda = lambda;
  end
  function setPhi(obj,phi)
    obj.phi = phi;
  end
  function setDamping(obj,damping)
    obj.damping = damping;
  end
  function setModeNrs(obj,modeNrs)
    obj.modeNrs = modeNrs;
  end
%
  % function scalePhi(obj,ScaleFactor)
  %     obj.phi = obj.phi*ScaleFactor;
  % end
%
  function data = clone(obj)
    data = ModalDataPkg.ModalData(obj.f,obj.lambda,obj.phi,obj.damping,obj.modeNrs);
  end
%
  % Note shapeInd can be negative, that can be understood that the
  % corresponding mode shape dimension will have a negative sign.
  function value = getExtracted(obj,modeInd,shapeInd)
    ind = find(shapeInd<0);
    newPhi = obj.phi;
    newPhi(abs(shapeInd(ind)),:) = -newPhi(abs(shapeInd(ind)),:);
    value = ModalDataPkg.ModalData(obj.f(modeInd),..., 
        obj.lambda(modeInd),..., 
        newPhi(abs(shapeInd),modeInd),..., 
        obj.damping(modeInd),modeInd);
  end
end
function reduceModeShapeDim(obj, ind)
    tmpInd = find(ind<0);
    obj.phi(abs(ind(tmpInd)),:) = -obj.phi(abs(ind(tmpInd)),:);
    setPhi(obj, obj.phi(abs(ind),:));
end

function reduceModeNrs(obj, ind)
    setFreq(obj, obj.f(ind));
    setLambda(obj, obj.lambda(ind));
    setPhi(obj, obj.phi(:,ind));
    setDamping(obj, obj.damping(ind));
    setModeNrs(obj, ind);
end

% function value = convertShape2FEModelXYZ(obj, feModel2ShapeXYZ)
% nDofsPerNode = length(feModel2ShapeXYZ);
% newPhi = obj.phi;
% value = ModalDataPkg.ModalData(obj.f(modeInd), . . .
% obj.lambda(modeInd), . . .
% newPhi(abs(shapeInd), modeInd), . . .
% obj.damping(modeInd), modeInd);
% value = [];
% end

function value = getModeShapeLength(obj)
    value = length(obj.phi);
end

% function [shapeHandles, meaPntHandles] = plot3Dstructure(obj, . . .
% geometry, cData, clrFlag, dataDesc, figNum)
% It is assumed that the data has 3 dofs per node
import UtilityPkg,*

feModel2ShapeXYZ = getFeModel2ShapeXYZ(geometry);
ii = find(feModel2ShapeXYZ);
nrOfDofsPerNode = find(ii);
upDir = getUpIndex( geometry);
%MeaNodeNrs = getMeaNodeNrs(Geometry);
elemNodeMap = getElemNodeMap( geometry);
nodeCoords = getNodeNr2Coords( geometry);

nrOfModes = length(obj.f);
%NrOfCols = 3;
%NrOfRows = ceil(NrOfModes/NrOfCols);

shapeHandles = zeros(nrOfModes,1);
meaPntHandles = zeros(nrOfModes,1);
% Scaling in such a way that maximum displacement will be
% scaled to 1/10 of the middle dimension
delta = max(nodeCoords(:,2:end))−min(nodeCoords(:,2:end));
deltaSorted = sort(delta);
deltaRef = deltaSorted(2);
if deltaRef == 0
    deltaRef = deltaSorted(3);
end

for k=1:nrOfModes
    figure(figNum(k));
    if clrFlag
        title([dataDesc ' mode ' num2str(obj.modeNrs(k)) ': ' . . .
            num2str(obj.f(k)) ' Hz '])
    else
        h=get(gca,'Title');
        t=get(h,'String');
        % set new title
        new_t=strcat(t,[': ' dataDesc ' mode ' . . .
            num2str(obj.modeNrs(k)) ': ' num2str(obj.f(k)) ' Hz ']);
        title(new_t);
    end

    delta = obj.phi(:,k);
    maxDispl = max(abs(delta));
```matlab
SF = deltaRef/(5*maxDisp1);
delta = reshape(delta,nrOfDofsPerNode,length(delta)/nrOfDofsPerNode)';
coords = nodeCoords(:,2:4);
coords(:,ii) = coords(:,ii) + SF*delta;

[~,shapeHandles(k),meaPntHandles(k)] = GeneralUtilities.PlotFunc(...
elemNodeMap,[nodeCoords(:,1) coords(:,1)],...
cData.upDir,getNodeNrs(geometry));
hold on
end;
end
end
```

### J.5.17 The ModePairingMethod class

**Source code**

```matlab
classdef ModePairingMethod < handle
methods (Abstract)
    pairModes(obj,modalData1,modalData2)
toString(obj)
end
end
```

### J.5.18 The ModeScalingMethod class

**Source code**

```matlab
classdef ModeScalingMethod < handle
methods (Abstract)
    scaleModes(obj,modalData1,modalData2)
toString(obj)
end
end
```

### J.5.19 The MSFResidual class

**Source code**

```matlab
classdef MSFResidual < ModalDataPkg.ResidMethod
methods
    % Input:
    % modalData1 – the modal data to scale
    % modalData2 – the reference modal data
    % Output:
    % column vector with signed scale factors for each mode shape from
    % the first modal data.
    scaleModes(obj,modalData1,modalData2)
toString(obj)
end
```
properties (Access = private)
    nrOfGroups
    sparseMtx % SEEMS TO BE NOT USED HERE!!
end

methods
    function obj = MSFResidual(nrOfGroups, sparseMtx)
        obj.nrOfGroups = nrOfGroups; % Number of (groups of) elements
        obj.sparseMtx = sparseMtx;
    end

    function [r_f r_s] = computeResid(obj, modalDataTest, modalDataFem)
        nuMea = getFreq(modalDataTest);
        phiMea = getPhi(modalDataTest);
        r_f = (nuSim.^2 - nuMea.^2) / nuMea.^2;
        r_s = bxsfun(@times, phiMea, dot(phiMea, phiSim, 1) ./ normPhiMea2) - phiSim;
    end

    function value = evaluateJacobian(obj, modalDataTest, modalDataFem, gradientData)
        phiFem = getPhi(modalDataFem);
        normPhiMea2 = sum(abs(phiMea).^2, 1);
        d_phim_d_ae = getD_phim_d_ae(gradientData);
        for ee = 1:obj.nrOfGroups
            JrLower(:,ee) = bxsfun(@times, phiMea, ...
                dot(phiMea, d_phim_d_ae(:,ee), 1) ./ normPhiMea2) ...
                - d_phim_d_ae(:,ee);
        end
    end

end

J.5.20 The MSFScaling class

Source code

classdef MSFScaling < ModalDataPkg.ModeScalingMethod
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    methods
        % Scale the first modal data with respect to the second
        function SignFactor = scaleModes(obj,modalData1,modalData2)
phi1 = getPhi(modalData1);
phi2 = getPhi(modalData2);

[n1,m1] = size(phi1);
[n2,m2] = size(phi2);

if (m1 ~= m2)
    error('Scaling failed: different number of mode shapes!');
else
    m = m1;
end

if (n1 ~= n2)
    error('Scaling failed: mode shapes have different lengths!');
end

MSF = zeros(m,1);
for i=1:m
    MSF(i) = (phi2(:,i).*phi1(:,i))/(phi1(:,i).*phi1(:,i));
end;

setPhi(modalData1,phi1*diag(MSF));
SignFactor = MSF;

function value=toString(obj)
    value = 'MSF';
end

J.5.21 The PointwiseWeighting class

Source code

classdef PointwiseWeighting < ModalDataPkg.WeightMethod
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    properties
        weights
    end

    methods
        function obj = PointwiseWeighting(freqWeights,shapeWeights)
            obj.weights = [ freqWeights(:); shapeWeights(:) ];
        end
        function obj = setWeights(weights)
            obj.weights = weights;
        end
        function value = getWeights(obj)
            value = obj.weights;
        end
        function value=toString(obj)
            value = 'Pointwise weighting of eigen frequencies and mode shapes';
        end
    end
end

J.5.22 The ReferenceBasedResidual class

Source code

classdef ReferenceBasedResidual < ModalDataPkg.ResidMethod
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    properties (Access = private)
        nrOfGroups
        sparseMtx % SEEMS TO BE NOT USED HERE!!
        resRefIndex
    end

function obj = ReferenceBasedResidual(nrOfGroups, sparseMtx)  
%obj.resRefIndex = resRefIndex;  
obj.nrOfGroups = nrOfGroups;  
% Number of (groups of) elements  
obj.sparseMtx = sparseMtx;  
end

function [r_f r_s] = computeResid(obj, modalDataTest, modalDataFem)  
nuMea = getNu(modalDataTest);  
nuSim = getNu(modalDataFem);  
phiMea = getPhi(modalDataTest);  
phiSim = getPhi(modalDataFem);  
r_f = (nuSim.^2 - nuMea.^2)/nuMea.^2;  
%phiSim = phiSim * diag(1 ./ phiSim(obj.resRefIndex,:));  
%phiMea = phiMea * diag(1 ./ phiMea(obj.resRefIndex,:));  
[r_f resRefIndex] = max(abs(phiSim));  
phiMea = phiMea * diag(1 ./ maxMea);  
phiSim = phiSim * diag(1 ./ maxSim);  
[r_s resRefIndex] = max(abs(phiSim));  
end

function value = evaluateJacobian(obj, modalDataTest, modalDataFem, gradientData)  
JrUpper = evaluateJrUpper(obj, modalDataTest, gradientData);  
JrLower = zeros(N*M, obj.nrOfGroups);  
d_phim_d_ae = getD_phim_d_ae(gradientData);  
for ee = 1:obj.nrOfGroups  
for mm = 1:M  
JrLower(:,:,ee,mm) = d_phim_d_ae(:,:,ee,mm) / phiSim(obj.resRefIndex(mm),mm);  
end
end

function value = toString(obj)  
value = 'Simple residual based on difference for mode shapes';  
end

classdef ResidMethod < handle  
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se  
methods (Abstract)
The SimpleResidual class

Source code

classdef SimpleResidual < ModalDataPkg.ResidMethod
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
properties (Access = private)
    nrOfGroups
    sparseMtx % SEEMS TO BE NOT USED HERE!!
    %nRef
end
methods
    function obj = SimpleResidual(nrOfGroups,sparseMtx)
        obj.nrOfGroups = nrOfGroups; % Number of (groups of) elements
        obj.sparseMtx = sparseMtx;
    end
    function [r_f r_s] = computeResid(obj,modalDataTest,modalDataFem)
        nuMea = getNu(modalDataTest);
        nuSim = getNu(modalDataFem);
        phiMea = getPhi(modalDataTest);
        phiSim = getPhi(modalDataFem);
        r_f = (nuSim.^2 - nuMea.^2)/nuMea.^2;
        r_s = phiSim(:) - phiMea(:);
        r_f = r_f(:); % Make column vector
        r_s = r_s(:); % Make column vector
        r = [r_f(:); r_s(:)]; % The residual
        disp('Freqs/Shapes')
        disp([sum(r_f.^2) sum(r_s.^2)])
    end
    function value = evaluateJacobian(obj,modalDataTest,modalDataFem,gradientData)
        JrUpper = evaluateJrUpper(obj,modalDataTest,gradientData);
        JrLower=zeros(N,M,obj.nrOfGroups);
        d_phim_d_ae = getD_phim_d_ae(gradientData);
        for ee=1:obj.nrOfGroups
            for mm=1:M
                JrLower(:,:,ee) = d_phim_d_ae(:,:,ee);
            end
        end
        JrLower=reshape(JrLower,N*M,obj.nrOfGroups);
        value = [JrUpper
                JrLower];
    end
    function value=toString(obj)
        value='Simple residual based on difference for mode shapes';
    end
end
J.5.25 The UniformFreqWeighting class

Source code

```matlab
classdef UniformFreqWeighting < ModalDataPkg.WeightingMethod
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    properties
        weights
    end

    methods
        function obj = UniformFreqWeighting(freqLen,shapesLen)
            obj.weights = [ ones(freqLen,1) zeros(shapesLen,1)];
        end
        function obj = setWeights(weights)
            obj.weights = weights;
        end
        function value = getWeights(obj)
            value = obj.weights;
        end
        function value=toString(obj)
            value='Uniform weighting of eigen frequencies and zero weighting of mode shapes';
        end
    end
end
```

J.5.26 The UniformShapeWeighting class

Source code

```matlab
classdef UniformShapeWeighting < ModalDataPkg.WeightingMethod
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    properties
        weights
    end

    methods
        function obj = UniformShapeWeighting(freqLen,shapesLen)
            obj.weights = [ zeros(freqLen,1) ones(shapesLen,1)];
        end
        function obj = setWeights(weights)
            obj.weights = weights;
        end
        function value = getWeights(obj)
            value = obj.weights;
        end
        function value=toString(obj)
            value='Uniform weighting of mode shapes and zero weighting of eigen frequencies';
        end
    end
end
```

J.5.27 The UniformWeighting class

Source code

```matlab
classdef UniformWeighting < ModalDataPkg.WeightingMethod
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    properties
```
J.5.28 The WeightingMethod class

Source code

```matlab
classdef WeightingMethod < handle
    %- Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se

    methods (Abstract)
        function obj = UniformWeighting(freqLen,shapesLen)
            obj.weights = [ ones(freqLen,1) ones(shapesLen,1) ];
        end
        function obj = setWeights(weights)
            obj.weights = weights;
        end
        function value = getWeights(obj)
            value = obj.weights;
        end
        function value=toString(obj)
            value='Uniform weighting of eigen frequencies and mode shapes';
        end
    end
end
```

J.6 OptimPkg package

The OptimPkg package for solving the optimisation problem. See also Section A.1 Consisting of the following classes.

J.6.1 The BoundConstraints class

Source code

```matlab
classdef BoundConstraints < OptimPkg.Constraints
    %- Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se

    properties (Access = private)
        lowerBound % lower bound constraints
        upperBound % upper bound constraints
    end

    methods
        function obj = BoundConstraints(lowerBound,upperBound)
            obj.lowerBound = lowerBound;
        end
```
J.6.2 The Constraints class

Source code

```matlab
classdef Constraints < handle
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    methods (Abstract)
    getValues(obj)
    end
end
```

J.6.3 The ElastModOptProblem class

Source code

```matlab
classdef ElastModOptProblem < OptimPkg.OptProblem
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    properties (Access = private)
    initParamValues
    constraints
    objectiveFunction
    end
    methods
    function obj = ElastModOptProblem(initParamValues,constraints,...
        objectiveFunction)
        obj.initParamValues = initParamValues;
        obj.constraints = constraints;
        obj.objectiveFunction = objectiveFunction;
    end
    function value = getConstraints(obj)
        value = obj.constraints;
    end
    function value = getInitParamValues(obj)
        value = obj.initParamValues;
    end
    function value = getObjectiveFunction(obj)
        value = obj.objectiveFunction;
    end
    end
end
```

J.6.4 The FemOptParamConvMethod class

Source code
J.6.5 The IBFemOptParamConv class

Interpolation based FEM to Optimization parameters converter with the following main methods.

Regularization with so-called shape functions, as suggested in [TMDR02] for beams. This method is a straightforward implementation of that technique with the suggestions in [SD+09] for generalization to shape functions defined on surfaces. It works when the updating parameters are physical parameters in point arranged in a rectangular grid or small perturbations of a rectangular grid. For other geometries or if one point is perturbed more than half-way to the next row or column in the grid, further generalizations are needed. See Section I.3 for a more detailed explanation and Section B.3 for applications of this interpolation on a concrete plate.

J.6.5.1 The method IBFemOptParamConv The line
\[
\text{obj.interpolFuns=} \text{InterpolFunctions(obj, meshPts(:,1:2), coarseMeshInd)}
\]
calls the method InterpolFunctions (Appendix J.6.5.5, page 199) for computing the matrix \(L\) with interpolation function samples at grid points stored as column vectors.

J.6.5.2 The method convertFem2OptCon

J.6.5.3 The method convertFem2Opt This function translates physical parameters \(P_{\rho}\) with initial value \(P_{\rho}^0\) to corresponding dimensionless optimization parameters \(p = \frac{P_{\rho}^0 - P_{\rho}}{P_{\rho}^0}\) as in (I.4).

J.6.5.4 The method convertOpt2Fem This method is the inverse of the function convertFem2Opt

\[
p = \frac{P_{\rho}^0 - P_{\rho}}{P_{\rho}^0} \quad \Leftrightarrow \quad P_{\rho} = P_{\rho}^0(1 - p),
\]

with \(p\) here renamed back to \(a\), as in (I.4).

The line \(a=\text{obj.interpolFuns*OptParams}\) computes the dimensionless updating parameter vector \(a\) via interpolation from parameters \(a^R\) computed for a coarse grid using equation (I.29), page 106, that is, \(a = La^R\). If no regularization of this type is done, then \(\text{obj.interpolFuns}\) is the identity matrix.

\*Rename a to p here for consistency with convertFem2Opt????????????

198
J.6.5.5 The method interpolFunctions This method computes the interpolating basis functions that are used for regularization, as explained in Section I.3.

The function call \( \mathbf{L} = \text{InterpolFunctions}(\text{obj}, \text{meshPts}, \text{coarseMeshInd}) \) returns the \( P \times R \) matrix \( \mathbf{L} \) that is defined in (I.28).

**Input parameters:**

- \( \text{meshPts} \) is a \( P \times 2 \)-matrix with the \( p \)th row containing the coordinates of the centre of mass of the group of elements corresponding to the \( p \)th updating parameter.
- \( \text{coarseMeshInd} \) is an index vector pointing out the rows in \( \text{meshPts} \) that correspond to points in the coarse mesh.

The rows in \( \text{meshPts} \) can be sorted in an arbitrary order. Therefore, the method \text{sortRectMeshPts} is called for arranging the mesh points into a (perturbed) rectangular grid:

J.6.5.6 The method sortRectMeshPts As described in Appendix I.3 and Figure I.2, page 105, the mesh points are assumed to be organized in a rectangular grid or a slight perturbation of a rectangular grid. The coarse mesh is required to be a rectangular subgrid (possibly empty or equal to the full grid). They are, however, not required to be sorted in any particular order in \( \text{meshPts} \). Thus the function \text{sortRectMeshPts} is used for sorting the mesh points into a rectangular grid with \( R \) rows and \( C \) columns.

**Output parameters:**

- \( \text{RectMesh} \) is a \( R \times C \times 2 \) array with \( \text{RectMesh}(r,c,:) \) being the coordinates of the parameter in the \( r \)th row and \( c \)th column.
- \( \text{RectMeshInd} \) is an index set such that \( \text{RectMesh}(r,c,d) = \text{meshPts}(\text{RectMeshInd}(r,c),d) \).
- \( \text{RowsInCoarseMesh} \) is an index vector telling which rows that are used for the coarse mesh.
- \( \text{ColsInCoarseMesh} \) is an index vector telling which columns that are used for the coarse mesh.

The method \text{sortRectMeshPts} should be able to find a (perturbed) rectangular grid as long all grid points are moved less than half way to a nearest neighbouring row or column.

J.6.5.7 The method zCoordsForPlane

**Source code**

```matlab
classdef IBFemOptParamConv < OptimPkg.FemOptParamConvMethod
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    % Interpolation based FEM to Optimization parameters converter
    properties
        coarseMeshInd
        interpolFuns
        femP0
        Pmin
        Pmax
    end
    methods (Access = public)
        function obj = IBFemOptParamConv(femP0,Pmin,Pmax,meshPts,coarseMeshInd)
            obj.femP0 = femP0;
            obj.Pmin = Pmin;
```

```
obj.Pmax = Pmax;
obj.coarseMeshInd = coarseMeshInd;
obj.interpFuns = InterpolFunctions(obj, meshPts(:,1:2), coarseMeshInd);
end

function OptCon = convertFem2OptCon(obj.femCon)
if isa(obj.femCon, 'OptimPkg.BoundConstraints')
    pMin = length(obj.coarseMeshInd); % == length(a)
    pMax = minFem(obj.coarseMeshInd)./obj.femP0(obj.coarseMeshInd).*ones(pLen, 1);
    OptCon = [OptimPkg.BoundConstraints(pMin, pMax);
else
    OptCon = [];
end

function p0 = convertFem2OptP0(obj)
p0 = 1−obj.femP0(obj.coarseMeshInd)./obj.femP0(obj.coarseMeshInd).*ones(pLen, 1);
end

function p = convertFem2Opt(obj, P)
plen = length(obj.coarseMeshInd);
p = 1−P(obj.coarseMeshInd , )./repeat(obj.femP0(obj.coarseMeshInd , ),1, size(P, 2)); ←
end

function P = convertOpt2Fem(obj, OptParams)
a = obj.interpFuns+OptParams;
P = obj.femP0.*(1−a);
end

function value = getInterpFuns(obj)
value = obj.interpFuns;
end

function value = getFemP0(obj)
value = obj.femP0;
end

function value = getPMin(obj)
value = obj.Pmin;
end

function value = getPMax(obj)
value = obj.Pmax;
end

end

methods (Access = private)
function L = InterpolFunctions(obj, meshPts, coarseMeshInd)
   % USAGE: L = InterpolFunctions(FineMesh, CoarseMeshRowColInd)

   % % INPUTS
   % % FineMesh = a 3-dimensional array with FineMesh(r,c, : ) and
   % % FineMesh(r,c, : ) being the x- and the y-coordinates
   % % of one point in the mesh
   % % CoarseMeshRowColInd = length 2 cell array with RowsInCoarseMesh and
   % % CoarseMesh being vectors containing the
   % % index sets for he first and second coordinate of
   % % FineMesh, respectively. This gives the coarse mesh
   % % for each point in which a tent function is computed.

   % % == If no input, then make a default choice for demonstration/testing ==
   if nargin==0
      DemoNode=true;
      NofRows=11;
      NofCols=31;
      NofRows=10 % DEBUG-OPTION !!!!
      NofCols=7 % DEBUG-OPTION !!!!
      D=2;
      SigmaMesh=0.1; % Standard deviation for random deviation of mesh points
                      % from the integers.
      SigmaMesh=0;
      FineMesh=zeros(NofRows, NofCols, D);
      for rr=1:NofRows
         for cc=1:NofCols
         end
   end
FineMesh(rr,cc,:)=[cc,rr];
end
end
FineMesh=FineMesh+SigmaMesh.*randn(size(FineMesh));
NofPtsFineMesh=NofRows*NofCols;
meshPts=reshape(FineMesh,NofPtsFineMesh,D);
ind=randperm(NofPtsFineMesh);
meshPts=meshPts(ind,:);

RowsInCoarseInd=[1:5:11];
ColsInCoarseInd=[1:5:31];

% DEBUG-OPTION !!!!
PtInCoarseGrid=remat(false,NofRows,NofCols);
PtInCoarseGrid(RowsInCoarseInd,ColsInCoarseInd)=true;
PtInCoarseGrid=PtInCoarseGrid();
% Reshape to column vector.
coarseMeshInd=find(PtInCoarseGrid);

else

DemoMode=false;
end

if ( length(meshPts(:,1)) == length(coarseMeshInd(:))
NofPtsFineMesh=length(meshPts(:,1));
L=eye(NofPtsFineMesh); % no interpolation
else

% == Sort mesh points into a (small deviation from) a rectangular mesh ==
RectMesh.RectMeshInd.RowsInCoarseMesh.ColsInCoarseMesh ...
= sortRectMeshPts(obj,meshPts,coarseMeshInd);
NofRows,NofCols,D=size(RectMesh);
NofPtsFineMesh=NofRows*NofCols;

% == Plot the numbering of the centerpoints ===============

if DemoMode
FigNr=0;
FigNr=FigNr+1;
figure(FigNr);
cif

FineGridColour=colour('green');
CoarseGridColour=colour('red');
xCoords=RectMesh(:,1);
xMin=min(xCoords);
xMax=max(xCoords);
yCoords=RectMesh(:,2);
yMin=min(yCoords);
yMax=max(yCoords);
delta=0.15;
axis([xMin-delta xMax+delta yMin-delta yMax+delta])

IsInCoarseGrid=remat([false],NofRows,NofCols);
IsInCoarseGrid(RowsInCoarseMesh,ColsInCoarseMesh)=true;
for cc=1:NofCols
for rr=1:NofRows
xy=RectMesh(rr,cc,:);
x=x+1;
y=y+2;
if IsInCoarseGrid(rr,cc)
text(x,y,int2str(RectMeshInd(rr,cc)),'BackgroundColor',CoarseGridColour)
else
text(x,y,int2str(RectMeshInd(rr,cc)),'BackgroundColor',FineGridColour)
end
end
end
hold on
end
hold off
xlabel('x')
ylabel('y')
title(['Center point positions (red for coarse mesh, \sigma-_mesh) = ' ...
num2str(SigmaMesh)])

num2str(SigmaMesh) ') numbered in the order that they appear in the input ←
\begin{verbatim}
J THE SOURCE CODE

parameter MeshPts.1) )

grid on

\%\% Create tent functions

NofCMrows=length(RowsInCoarseMesh);
NofCMcols=length(ColsInCoarseMesh);

L=zeros(NofPtsFineMesh,NofCMrows+NofCMcols);
%TentFctNr=0;
for cInd=1:NofCMcols
    cc=ColsInCoarseMesh(cInd);
    for rInd=1:NofCMrows
        %TentFctNr=TentFctNr+1;
        rr=RowsInCoarseMesh(rInd);
        TentFct=zeros(NofRows,NofCols);
        TentFct(rr,cc)=1;
        ind=max(find(RowsInCoarseMesh<rr));
        if length(ind)==0
            PrevRow=rr;
        else
            PrevRow=RowsInCoarseMesh(ind);
        end
        ind=min(find(RowsInCoarseMesh>rr));
        if length(ind)==0
            NextRow=rr;
        else
            NextRow=RowsInCoarseMesh(ind);
        end
        ind=max(find(ColsInCoarseMesh<cc));
        if length(ind)==0
            PrevCol=cc;
        else
            PrevCol=ColsInCoarseMesh(ind);
        end
        ind=min(find(ColsInCoarseMesh>cc));
        if length(ind)==0
            NextCol=cc;
        else
            NextCol=ColsInCoarseMesh(ind);
        end

        \%\% Create all 1-8 "tent walls"

        \% Lower left quadrant:
        rInd=PrevRow:rr;
        cInd=PrevCol:cc;
        if ( (length(rInd)>1) \&\& (length(cInd)>1) )
            x = RectMesh(rInd,cInd,1);
            y = RectMesh(rInd,cInd,2);
            PointsInPlane=[x(end,end) x(1,1) x(1,end)
                           y(end,end) y(1,1) y(1,end)];
            TentWall1=zCoordsForPlane(obj,PointsInPlane,x,y);
            TentWall2=zCoordsForPlane(obj,PointsInPlane,x,y);
            TentFct(rInd,cInd)=min(TentWall1,TentWall2);
        end

        \% Upper left quadrant:
        rInd=rr:NextRow;
        cInd=PrevCol:cc;
        if ( (length(rInd)>1) \&\& (length(cInd)>1) )
            x = RectMesh(rInd,cInd,1);
            y = RectMesh(rInd,cInd,2);
            PointsInPlane=[x(end,end) x(1,end) x(1,1)
                           y(end,end) y(1,end) y(1,1)];
            TentWall1=zCoordsForPlane(obj,PointsInPlane,x,y);
            TentWall2=zCoordsForPlane(obj,PointsInPlane,x,y);
            TentFct(rInd,cInd)=min(TentWall1,TentWall2);
        end
\end{verbatim}
y(end,1) y(1,end) y(end,end)
0 1 0

TentWall2=zCoordsForPlane(obj,PointsInPlane,x,y);
TentFct(rInd,cInd)=min(TentWall1,TentWall2);
end

% Upper right
rInd=rr; NextCol;
cInd=cc; NextRow;
if ( (length(rInd)>1) && (length(cInd)>1) )
x = RectMesh(rInd,cInd,1);
y = RectMesh(rInd,cInd,2);
PointsInPlane=[x(1,1) x(end,1) x(end,end)]
y(1,1) y(end,1) y(end,end)
1 0 0]
TentWall1=zCoordsForPlane(obj,PointsInPlane,x,y);
PointsInPlane=[x(1,1) x(end,end) x(1,end)]
y(1,1) y(end,end) y(1,end)
0 1 0]
TentWall2=zCoordsForPlane(obj,PointsInPlane,x,y);
TentFct(rInd,cInd)=min(TentWall1,TentWall2);
end

% lower right
rInd=PrevRow;rr;
cInd=cc; NextCol;
if ( (length(rInd)>1) && (length(cInd)>1) )
x = RectMesh(rInd,cInd,1);
y = RectMesh(rInd,cInd,2);
PointsInPlane=[x(end,1) x(1,1) x(1,end)]
y(end,1) y(1,1) y(1,end)
1 0 0];
TentWall1=zCoordsForPlane(obj,PointsInPlane,x,y);
PointsInPlane=[x(end,end) x(1,end) x(1,1)]
y(end,end) y(1,end) y(1,1)
0 0 1];
TentWall2=zCoordsForPlane(obj,PointsInPlane,x,y);
TentFct(rInd,cInd)=min(TentWall1,TentWall2);
end

if DemoMode
FigNr=FigNr+1;
figure(FigNr);
x = RectMesh(:, :, 1);
y = RectMesh(:, :, 2);
% mesh(1: NofCols , 1: NofRows, L);
surf(x,y,TentFct);
xlabel('x')
ylabel('y')
title(['"Tent function with center point (' ...
num2str(RectMesh(rr,cc,1)) ', ' ...
num2str(RectMesh(rr,cc,2)) ')']
end

% Finally, reorder the tent function points in the original order given
% by the index set RectMeshInd and the relation
% for rrr=1:NofRows
for ccc=1:NofCols
  % Fine mesh point numbers for the current tent function:
  CurrentTentFctFineMeshPtNr=RectMeshInd(rr,cc);
  TentFctNr=find( coarseMeshInd==CurrentTentFctFineMeshPtNr );
  %RowCol=MeshPtRowColNr(coarseMeshInd(TentFctNr));
  % TentFctColNr=RectMeshInd(coarseMeshInd(TentFctNr));
  L(RectMeshInd(rrr,ccc), TentFctNr)=TentFct(rrr,ccc);
end
end

% MeshPtRowColNr=zeros(NofPtsFineMesh, 2);
% for nn=1:NofPtsFineMesh
% [RowInd, ColInd] = (findRectMeshInd==nn);
% MeshPtRowColNr(RowInd, ColInd) = nn;
 end

function [RectMesh, RectMeshInd, RowsInCoarseMesh, ColsInCoarseMesh] ...
    = sortRectMeshPts(obj, meshPts, coarseMeshInd)

% INPUT
% MeshPts = Nx2--matrix with each row containing the x-- and
% y-coordinate of one point in the mesh
% coarseMeshInd = vector containing the row indices of the points in the
% mesh that belongs to the coarse mesh.
% OUTPUTS
% RectMesh = the mesh points sorted into a rectangular array
%    RectMesh(m,n,d), with smaller m (or n)
%    corresponding to smaller y (or x) and with
%    d=1 and d=2 giving the x and the y-coordinate,
%    respectively.
% RectMeshInd = index set such that
%    RectMesh(m,n,d) = MeshPts(RectMeshInd(m,n),d)
% RowsInCoarseMesh = vector telling what rows that are used for the
% coarse mesh
% ColsInCoarseMesh = vector telling what columns that are used for the
% coarse mesh

% Reorder the mesh points in increasing x-coordinate order
[MeshPtsSortedX, indX] = sort(meshPts(:,1)) ;
MeshPtsSortedX = meshPts(indX, :);

% Find the number of rows such that the mesh point positions is a small
% deviation from a rectangular mesh of size NofRows x NofCols:
NofPts = length(meshPts(:,1));
PossibleNofRows = divisors(obj, NofPts) ; % All possible column lengths
PossibleNofRows = PossibleNofRows(2:end-1) ; ' ; % Look for rectangular mesh

% Default if no rectangular mesh is found:
RectMesh = MeshPtsSortedX ;
indY = (1:NofPts) ; ' ;
RectMeshFound = false ;

for NofRows = PossibleNofRows ;
    NofCols = NofPts/NofRows ;

    % Reshape before sorting:
    MeshPtsSortedXY = reshape(MeshPtsSortedX, NofRows, NofCols, 2) ;
    % Sort each column in increasing y-coordinate order
    [MeshPtsSortedXY(:, :, 2), indY] = sort(MeshPtsSortedXY(:, :, 2)) ;
    for cc = 1:NofCols
        MeshPtsSortedXY(:, cc, 1) = MeshPtsSortedXY(indY(:, cc), cc, 1) ;
    end

    % Check if the current value of NofRows gives a separation of rows & cols
    NotRectMesh = false ;
    for rr = 1:NofRows-1
        % Check if each row is located below next row:
        if max(MeshPtsSortedXY(rr, :, 2)) >= min(MeshPtsSortedXY(rr+1, :, 2))
            NotRectMesh = true ;
        end
    end
    for cc = 1:NofCols-1
        % Check if each column is located to the left of next column:
        if max(MeshPtsSortedXY(:, cc)) >= min(MeshPtsSortedXY(:, cc+1))
            NotRectMesh = true ;
        end
    end

    if ~NotRectMesh
        RectMeshFound = true ;
        RectMesh = MeshPtsSortedXY ;
        RectMeshInd = reshape(indX, NofRows, NofCols) ;
        for cc = 1:NofCols
            RectMeshInd(:, cc) = RectMeshInd(indY(:, cc), cc) ;
        end
    end
if ~RectMeshFound
    figure(666)
    clf
    for nn=1:length(meshPts)
        plot(meshPts(nn,1),MeshPts(nn,2),'rs')
    hold on
    end
    hold off
    error(['The input mesh points are not positioned with small ' ...
           'deviations from a rectangular mesh'])
end

% Find rows and columns used for the coarse mesh ---------------
[NofRows,NofCols]=size(RectMeshInd);
blah=zeros(NofRows,NofCols);
for nn=1:length(coarseMeshInd(:))
    ind=find(RectMeshInd==coarseMeshInd(nn));
    blah(ind)=1;
end
% If there are two different nonzero rows or columns then the rank >1:
if rank(blah)>1
    error('not possible to build tent functions for this mesh')
end
RowsInCoarseMesh = find( sum(blah,2) ~=0 );
ColsInCoarseMesh = find( sum(blah),2 ~=0 );

function nVec = divisors(obj,N)
% INPUT
% N = an integer
% OUTPUT
% nVec = a column vector containing all integers dividing N, stored in
% increasing order.
PrimeFactors=factor(N);
PrimeFactorsPowers=findPowers(obj,PrimeFactors); % Defined below
nVec = combFactors(obj,PrimeFactorsPowers);
end

function PrimeFactorsPowers=findPowers(obj,primeFactors)
% INPUTS
% PrimeFactors = a vector containing a sequence of prime factors in
% increasing order
% OUTPUTS
% PrimeFactors = a 2 column vector with the first column containing the
% unique prime factors in PrimeFactors and the second
% column containing the number of occurrences of that prime
% factor in PrimeFactors.
ind=find(primeFactors==primeFactors(1));
pow=length(ind);
if pow==length(primeFactors)
    PrimeFactorsPowers=[primeFactors(1) pow];
else
    PrimeFactorsPowers=[primeFactors(1) pow
                        findPowers(obj,primeFactors(pow+1:end))];
end
end

function factors = combFactors(obj,primeFactorsPowers)
    if length(primeFactorsPowers(:,1))==1
        tmp=1;
    else
        tmp=combFactors(obj,primeFactorsPowers(2:end,:));
    end
NextPower=primeFactorsPowers(1,2);
factors=zeros((NextPower+1)*length(tmp),1);
for pp = 0:NextPower % The different powers
```matlab
factors(pp*length(tmp)+(1:length(tmp))) = ... 
[primeFactorsPowers(1,1)' pp*tmp ];
end
factors = sort(factors);
end
function z = zCoordsForPlane(obj, pointsInPlane, x, y)
    % For a specified plane P, this function computes the z-coordinates for
    % points with x- and y-coordinates
    % INPUTS
    % PointsInPlane = 3x3 matrix with column vectors giving the location for
    % three points in the plane P.
    % x, y = numbers or arrays of the same size.
    % OUTPUTS
    % z = number or array the same size as x and y, with z(m,n,...) being the
    % z-coordinate for the point on the plane with x- and y-coordinates
    % x(m,n,...) and y(m,n,...), respectively.
    % Compute normal vector for the plane
    a = pointsInPlane(:,1);
    b = pointsInPlane(:,2);
    c = pointsInPlane(:,3);
    n = cross(b-a, c-a);
    if n(3) == 0
        error('Plane parallel with z-axis. No uniquely determined z-coordinate. ')
    end
    % Standard formula for points in the plane
    z = -( (x-a(1))*n(1) + (y-a(2))*n(2) )/n(3) + a(3);
end
end
```

### J.6.6 The NTROptimizer class

**Source code**

```matlab
classdef NTROptimizer < OptimPkg.Optimizer
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    properties
        optProblem
        groupGeometry % for showing updating data
        modeShapeGeometry % for showing modal data
        iterationDataStorage % after each iteration the iteration data
        % will be stored here
        outfun % to be run after each iteration
        femOptParamConv
    end
    methods
        function obj = NTROptimizer(optProblem, groupGeometry, ...
            modeShapeGeometry, ...
            iterationDataStorage, ...
            outfun)
            obj.optProblem = optProblem;
            obj.groupGeometry = groupGeometry;
            obj.modeShapeGeometry = modeShapeGeometry;
            obj.iterationDataStorage = iterationDataStorage;
            obj.femOptParamConv = getFem2OptParamConv(getObjectiveFunction(...
                optProblem));
            outfun0 = @(p,optimValues,state) saveIterationData(obj,p,optimValues,state);
            if isempty(outfun)
                obj.outfun = [{outfun0} outfun];
            else
```
function [p,fval] = optimize(obj)
    ObjectiveFunction = getObjectiveFunction(obj.optProblem);
    %Conv = getFem2OptParamConv(ObjectiveFunction);
    % OutFunObj = OptimPkg.OutFunClass(getModalDataStorage(ObjectiveFunction), ... 
    %   obj.groupGeometry, ... 
    %   obj.modeShapeGeometry, ... 
    %   Conv);

    %if ~isempty(obj.outFun)
    outfun1 = @(p,optimValues,state) plotUpdatData(obj.outFun1,p,optimValues,←
    %  state,1); 
    outfun2 = @(p,optimValues,state) plotModalData(obj.outFun2,p,optimValues,←
    %  state); 
    outfun3 = @(p,optimValues,state) plotObjFunValue(obj.outFun3,p,optimValues,←
    %  state,200); 
    %outfun = {outfun0, outfun1, outfun2};
    %outfun = {outfun0, outfun1, outfun2, outfun3};
    % else
    outfun = outfun0;
    % end

    options = optimset('Algorithm','trust-region-reflective',... 
    'GradObj','on',... 
    'Hessian','user-supplied',... 
    'Display','iter',... 
    'OutputFcn',obj.outFun);
    %options = optimset('Algorithm','trust-region-reflective',... 
    %'GradObj','on',... 
    %'Hessian','user-supplied',... 
    %'Display','iter',... 
    %'OutputFcn',obj.outFun);
    %warning('Should consider another flexible implementation for out function');

    %ObjectiveFunction = @(p) ObjFun(p OptCon, StructUpdating);
    ObjFun = @(p) evaluate(ObjectiveFunction,p);
    p0 = getInitParamValues(obj.optProblem);
    [A,b,Aeq,beq,lb,ub,nonlcon] = obj.optProblem.getConstraints().getValues();
    %[p,fval] = fmincon(ObjFun,p0,A,b,Aeq,beq,lb,ub,nonlcon,options);
    [p,f] = fmincon(ObjFun,p0,A,b,Aeq,beq,lb,ub,nonlcon,options);
end

function stop = saveIterationData(obj,p,optimValues,state)
    stop = false;
    switch state
    case 'init'
        %hold on
    case 'iter'
        %iteration = optimValues.iteration;
        % save the optimization data into file numbered by iteration
        % save the updated parameters into file numbered by iteration
        %disp(['ITERATION=',num2str(iteration)])
        %disp('OPTIMVALUES')
        %disp(optimValues)
        %disp('STATE')
        %disp(state)
        P = convertOpt2Fem(obj.femOptParamConv,p); % updating parameters
        f = optimValues.fval; % objective function value
        storeParameters(obj.iterationDataStorage,P,f,optimValues);
        case 'done'
            %hold off
        otherwise
            %hold on
        end
end

function value=toString(obj)
    value='Newton Trust Region optimization';
J.6.7 The ObjectiveFunction class

J.6.7.1 The method evaluate  The method evaluate computes the objective function, its gradient and hessian.

The line \( \mathbf{J}_{rc} = \mathbf{J}_{ra} \cdot \text{getInterpolFuns}(\text{obj.fem2OptParamConv}) \) computes the Jacobian for the updating parameters via equation (I.30), page 106: \( \mathbf{J}_{rc} = \mathbf{J}_{ra} \). Then the line \( \mathbf{\nabla f} = \mathbf{J}_{rc}^T \mathbf{W} \mathbf{r} \) computes the corresponding gradient using equation (I.12) and finally, the line \( \mathbf{Hessf} = \mathbf{J}_{rc}^T \mathbf{W} \mathbf{J}_{rc} \) computes the Hessian from the approximation (I.14).

Source code

```matlab
classdef ObjectiveFunction
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    properties (Access = private)
        simulator
        gradient
        residual
        modePairing
        modeScaling
        weighting
        fem2OptParamConv
        testModalData
        iterationDataStorage
        femModelInd
        testModelInd
        phiPlotSubInd
        phiMeaFemInd
    end

    methods
        function obj = ObjectiveFunction(simulator,...
            iterationDataStorage,...
            gradient,residual,...
            modePairing,...
            modeScaling,...
            weighting,...
            fem2OptParamConv,...
            testModalData,...
            testModelInd,femModelInd,...
            phiPlotSubInd,...
            phiMeaFemInd)
            obj.simulator = simulator;
            obj.gradient = gradient;
            obj.residual = residual;
            obj.testModalData = testModalData;
            obj.modePairing = modePairing;
            obj.modeScaling = modeScaling;
            obj.weighting = weighting;
            obj.fem2OptParamConv = fem2OptParamConv;
            obj.femModelInd = femModelInd;
            obj.testModelInd = testModelInd;
            obj.phiPlotSubInd = phiPlotSubInd;
            obj.phiMeaFemInd = phiMeaFemInd;
            obj.iterationDataStorage = iterationDataStorage;
        end

        function value = getFem2OptParamConv(obj)
            value = obj.fem2OptParamConv;
        end
```
% were paired it is needed to change the weighting accordingly!
% (in increasing order of frequencies) of test
% mode shapes and since the order was changed after mode shapes
% Since weighting was defined for the initial ordering

function [f, gradf, hessf] = evaluate(obj, params)
    import SimulatorPkg.ResultType
    import ModalDataPkg.*
    %a = OptConst.NDamageFcts*params;
    P = convertOpt2Fem(obj.fem2OptParamConv, params);
    update(obj.simulator, P, ResultType.MODAL_DATA);
    run(obj.simulator, ResultType.MODAL_DATA);
    plottedModalDataFem = getResult(obj.simulator, ResultType.MODAL_DATA);
    %
    % TEMP_TO_REMOVE = getResult(obj.simulator, ResultType.MODAL_DATA);
    % modes = getPhi(TEMP_TO_REMOVE);
    % nrofModes = size(modes, 2);
    for i=1:nrofModes
        m = modes(:,i);
        plots = m(obj.femModelInd);
        any(plotm(obj.phiPlotSubInd) == m(obj.phiMeaFemInd))
    end
    %
    GD = evaluateGradient(obj, gradient, plottedModalDataFem);
    % make a copy of the test modal data
    plottedModalDataTest = clone(obj.testModalData);
    % reduce test and FEM mode shapes dimensions to the plotted dimensions
    reduceModeShapeDim(plottedModalDataFem, obj.femModelInd);
    reduceModeShapeDim(plottedModalDataTest, obj.testModelInd);
    % pair mode shapes with reduced to the plotted dimensions
    test2FemModeIndMap = pairModes(obj.modePairing, ...
        plottedModalDataFem, plottedModalDataTest);
    % reorder modal data with respect to their similarity
    reduceNrs(plottedModalDataFem, test2FemModeIndMap(:,2));
    reduceNrs(plottedModalDataTest, test2FemModeIndMap(:,1));
    % scale mode shapes
    scaleModes(obj.modeScaling, plottedModalDataTest, plottedModalDataFem);
    % reduce gradient to measured dimensions
    GD = getExtracted(GD, test2FemModeIndMap(:,2), obj.phiMeaFemInd);
    % reduce reduced (plotted) mode shapes to measured dimensions
    ModeNr = 1:length(test2FemModeIndMap);
    resModalDataTest = getExtracted(plottedModalDataTest, ModeNr, obj.phiPlotSubInd);
    resModalDataFem = getExtracted(plottedModalDataFem, ModeNr, obj.phiPlotSubInd);
    %
    % Figure
    %
    testM = getPhi(resModalDataTest);
    femM = getPhi(resModalDataFem);
    NN = size(testM, 2);
    for kk=1:NN
        %
        % Figure
        %
        % test = testM(:,kk);
        % plot(test,’g’);
        % fem = femM(:,kk);
        % hold on
        % plot(fem,’b’)
        % end
    %
    % compute residual
    [r_f, r_m] = computeResid(obj.residual, resModalDataTest, resModalDataFem);
    r = [r_f(:,1); r_m(:,1)];
    weights = getWeights(obj.weighting);
    % Since weighting was defined for the initial ordering
    % (in increasing order of frequencies) of test
    % mode shapes and since the order was changed after mode shapes
    % were paired it is needed to change the weighting accordingly!
\begin{verbatim}
ind = length(test2FemModeIndMap)+1:length(weights);
n = length(getPhi(resModalDataFem));
% transform mode shapes into matrix form
shapeWeights = reshape(weights(ind),n,length(ind)/n);
% reorder mode shapes
shapeWeights = shapeWeights(:,test2FemModeIndMap(:,1));
weights = [weights(test2FemModeIndMap(:,1)); shapeWeights(:)];
W = diag(weights); % Diagonal weight matrix
f = sum(weights'*r.^2)/2; % The objective function
% extract appropriate data from big GD and MDfem
Jra = evaluateJacobian(obj.residual,...
    resModalDataTest,resModalDataFem,GD);
Jrp = Jra*getInterpolFuns(obj.fem2OptParamConv);
graf = Jrp.'*W*r;
hessf = Jrp.'*W*Jrp;
storeModalData(obj.iterationDataStorage,plottedModalDataTest,...
    plottedModalDataFem);
end
end
end

J.6.8 The Optimizer class
Source code

\begin{verbatim}
classdef Optimizer < handle
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    % properties
    % OptProblem
    % Simulator
    methods (Abstract)
        %setOptProblem(obj)
        %setSimulator(obj)
        optimize(obj)
        toString(obj)
    end
end
\end{verbatim}

J.6.9 The OptProblem class
Source code

\begin{verbatim}
classdef OptProblem < handle
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    methods (Abstract)
        getConstraints(obj)
        getInitParamValues(obj)
       getObjectiveFunction(obj)
    end
end
\end{verbatim}

J.6.10 The OutFunClass class
Source code
classdef OutFunClass
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se

properties (Access = private)
iterationDataStorage
   groupGeometry % Description of group geometry in FEM coordinate system
   modeShapeGeometry % Description of test geometry in FEM coordinate system
   femOptParamConv
   figNum
   modeShapeMeaFemHandles
   meaPointHandles
end

methods
   function obj = OutFunClass(iterationDataStorage ,... 
       groupGeometry ,modeShapeGeometry ,... 
       femOptParamConv)
       obj.iterationDataStorage = iterationDataStorage;
       obj.groupGeometry = groupGeometry;
       obj.modeShapeGeometry = modeShapeGeometry;
       obj.femOptParamConv = femOptParamConv;
       obj.modeShapeMeaFemHandles = [];
       obj.meaPointHandles = [];
       obj.figNum = [];
   end

   function stop = plotModalData(obj,p,optimValues,state)
import UtilityPkg ,*
   %iteration = optimValues.iteration;
   stop = false;
   switch state
       case 'init'
           ModalDataMea = getModalDataMea(obj.iterationDataStorage);
           ModalDataSim = getModalDataSim(obj.iterationDataStorage);
           n = length(getNu(ModalDataMea));
           %initializeHandlers(obj,NrOfModes);
           if isempty(obj.modeShapeMeaFemHandles)
               obj.modeShapeMeaFemHandles = zeros(n,2);
           end
           if isempty(obj.meaPointHandles)
               obj.meaPointHandles = zeros(n,2);
           end
           if isempty(obj.figNum)
               obj.figNum = 2:(n+1);
           end
           if ~isempty(ModalDataMea) && ~isempty(ModalDataSim)
           if (any(any(obj.modeShapeMeaFemHandles)==0))
               for i=1:n
                   % Remove old plot of Mea & FEM mode shapes
                   figure(obj.figNum(i));
                   delete(obj.modeShapeMeaFemHandles(i,1));
                   delete(obj.modeShapeMeaFemHandles(i,2));
                   delete(obj.meaPointHandles(i,1));
                   delete(obj.meaPointHandles(i,2));
               end
           end
           else
               for i=1:n
                   if ~isempty(find(obj.'type','figure')==obj.figNum(i,1))
                       clf(obj.figNum(i));
                   end
               end
           figure(obj.figNum(i))
           GeneralUtilities.PlotFunc (... 
              getElemNodeMap(obj.modeShapeGeometry) ,... 
              getNodeNr2Coords(obj.modeShapeGeometry) ,... 
              'g' ,getUpIndex(obj.modeShapeGeometry) ,... 
              getMeaNodeNrs(obj.modeShapeGeometry));
       end

end
% handlesMea, handlesMeaTestPnts = plot3Dstructure(ModalDataMea,...
    obj.modeShapeGeometry,'r',true,'Mea',obj.figNum);
% handlesFem, handlesMeaFemPnts = plot3Dstructure(ModalDataSim,...
    obj.modeShapeGeometry, 'b',false,'Sim',obj.figNum);
% if (all(all((obj.modeShapeMeaFemHandles==0)))
    obj.modeShapeMeaFemHandles = [handlesMea handlesFem];
    obj.meaPointHandles = [handlesMeaTestPnts handlesMeaFemPnts];
end

end

if (all(all(obj.meaPointHandles==0))
    obj.meaPointHandles = handlesMeaTestPnts handlesMeaFemPnts;
end

end

%if (all(all((obj.modeShapeMeaFemHandles))

function stop = plotUpdatData(obj,p,optimValues,state,FigNum)
stop = false;
switch state
    case 'init'
        hold on
    case 'iter'
        P = convertOpt2Fem(obj.femOptParamConv,p);
        plotUpdatSimData(obj,P,{'E0','E{updated}'},FigNum);
    case 'done'
        hold off
    otherwise
        hold on
end

function stop = plot2ParamsBasedObjFun(obj,p,optimValues,state,figNum)
stop = false;
switch state
    case 'init'
        hold on
    case 'iter'
        P = convertOpt2Fem(obj.femOptParamConv,p);
        if length(p)=2
            error('Function is only for plotting objective function based on two parameters!')
        end
        figure(figNum)
        plot3(P(1),P(2),optimValues.fval,'r*')
        hold on
    case 'done'
        hold off
    otherwise
        hold on
end

methods (Access = private)

function obj = initializeHandlers(obj,nrOfModes)
    if isempty(obj.modeShapeMeaFemHandles)
        obj.modeShapeMeaFemHandles = zeros(nrOfModes,2);
    end
    if isempty(obj.meaPointHandles)
        obj.meaPointHandles = zeros(nrOfModes,2);
    end
    if isempty(obj.figNum)
        obj.figNum = 2:(nrOfModes+1);
    end

function plotUpdatSimData(obj,P,Title,FigNum)
import UtilityPkg.*

% INPUTS
% GroupBoundPts = Cell array containing 2-column matrices.
% The column vectors of each matrix contains x- and y-coordinates of the corner points of a filled polygon
SimulatorPkg package

The SimulatorPkg interface for solving the FEM generalized eigenvalue problem, as described in Section A.1.

J.7.1 The ResultType class

Source code

```matlab
if ~isempty(find(findobj('type','figure')==FigNum,1))
clf(FigNum);
end;

FemUpInd = getUpIndex(obj.groupGeometry);
GroupElemNodeMap = getElemNodeMap(obj.groupGeometry);
NodeCoords = getNodeNr2Coords(obj.groupGeometry);
MeaNodes = getMeaNodeNrs(obj.groupGeometry);
P0 = getFemP0(obj.femOptParamConv);
Pmin = getPmin(obj.femOptParamConv);
Pmax = getPmax(obj.femOptParamConv);
figure(FigNum)
PlotFunc (GroupElemNodeMap, NodeNrCoords, P, FemUpInd, getMeaNodeNrs(obj.modeShapeGeometry), 'none', true);
end

function [xCoords, yCoords]=cell2mat(obj, CornerPts)
NrOfPolygons=length(CornerPts);
MaxNoOfCorners=0;
for nn=1:NrOfPolygons
    MaxNoOfCorners=max(MaxNoOfCorners, length(CornerPts{nn}(:,1)));
end
xCoords=zeros(MaxNoOfCorners, NrOfPolygons);
yCoords=zeros(MaxNoOfCorners, NrOfPolygons);
for PolyNr=1:NrOfPolygons
    CornerPtsX=CornerPts{PolyNr}(:,1);
    CornerPtsY=CornerPts{PolyNr}(:,2);
    NofCorners=length(CornerPtsX);
    xCoords(1:NofCorners, PolyNr)=CornerPtsX;
    yCoords(1:NofCorners, PolyNr)=CornerPtsY;
end
end
end
```
classdef ResultType
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
enumeration
MODAL_DATA
GROUP_STIFF_MTX
GLOBAL_NODE_COORDS
GROUP_ELEM_NODE_NRS
GROUP_CENTER_COORDS
end
end

J.7.2 The Simulator class

Source code

classdef Simulator < handle
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
methods (Abstract)
% Updates the input file corresponding to the required result type for the simulator with the given parameter
% Input:
% parameterArray – array of parameter values
% resultType – type of the required result for simulation
update(obj, parameterArray, resultType)
% Why ResultType here?
% Runs the simulator to get the required result type.
% Input:
% resultType – type of the required result for simulation
run(obj, resultType)
% Gets the simulation result.
% Input:
% resultType – type of the required result
getResult(obj, resultType)
% Cleans temporary files and data structures.
clear(obj)
end

J.8 The UtilityPkg package

J.8.1 The dlnode class

Contains the methods dlnode, insertAfter, insertBefore, disconnect, delete and disp for maintaining doubly linked lists.

Source code

classdef dlnode < handle
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
% dlnode A class to represent a doubly-linked node.
% Link multiple dlnode objects together to create linked lists.
% When you create a linked list and assign a variable that contains,
% for example, the head or tail of the list, clearing that variable
% causes the destructor to recurse through the entire list. With large
% enough list, clearing the list variable can result in MATLAB
% exceeding its recursion limit.

% The clearList method avoids recursion and improves the performance
% of deleting large lists by looping over the list and disconnecting
% each node. clearList accepts the handle of any node in the list and
% removes the remaining nodes.

properties
  Data
end

properties(SetAccess = private)
  Next% OLD: = dlnode.empty;
  Prev% OLD: = dlnode.empty;
end

methods
  function node = dlnode(Data)
    % Construct a dlnode object
    if nargin > 0
      node.Data = Data;
    end
  end

  function insertAfter(newNode, nodeBefore)
    % Insert newNode after nodeBefore.
    removeNode(newNode);
    newNode.Next = nodeBefore.Next;
    newNode.Prev = nodeBefore;
    if ~isempty(nodeBefore.Next)
      nodeBefore.Next.Prev = newNode;
    end
    nodeBefore.Next = newNode;
  end

  function insertBefore(newNode, nodeAfter)
    % Insert newNode before nodeAfter.
    removeNode(newNode);
    newNode.Next = nodeAfter;
    newNode.Prev = nodeAfter.Prev;
    if ~isempty(nodeAfter.Prev)
      nodeAfter.Prev.Next = newNode;
    end
    nodeAfter.Prev = newNode;
  end

  function removeNode(node)
    % Remove a node from a linked list.
    if ~isscalar(node)
      error('Input must be scalar')
    end
    prevNode = node.Prev;
    nextNode = node.Next;
    if ~isempty(prevNode)
      prevNode.Next = nextNode;
    end
    if ~isempty(nextNode)
      nextNode.Prev = prevNode;
    end
    node.Next = [];% OLD dlnode.empty;
    node.Prev = [];% OLD dlnode.empty;
  end

  function clearList(node)
    % Clear the list before
    % clearing list variable
    % Avoid recursive calls to destructor as a result of clearing the list
    % variable. Loop through list to disconnect each node. When there are
    % no references to a node, MATLAB calls the class destructor (see the
    % delete method) before deleting it.
J.8.2 The GeneralUtilities class

Source code  Here follows first the full source code, and then a short description of some of the commands contained in it.

classdef GeneralUtilities
% Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
methods (Static)
% === Convert a cell array of strings to a row vector of numbers ====
function arr = cell2num(cellArray)
    % Avoid empty string caused by trailing commas:
    if length(cellArray{end})==0
        cellArray=cellArray(1:end-1);
    end
    len = length(cellArray);
    arr = zeros(1,len);
    for i=1:len
        % NextStr=cellArray{i};
        % if ischar(NextStr)
        arr(i) = str2num(cellArray{i});
        % end
    end;
end

function S=Triang2SymmMtx(T)
    S = T + (T-diag(diag(T))).';
end

function [row, col] = FindCells(string,CellArray)
    cellfind = @(string}@{cell_contents}(strcmp(string,cell_contents));
    cells = cellfun(cellfind,string).CellArray);
    [row, col] = find(cells);
end

% ignore case = case insensitive
function [row, col] = FindCellsI(string,CellArray)
    cellfind = @(string}@{cell_contents}(strcmpi(string,cell_contents));
    cells = cellfun(cellfind,string).CellArray);
    [row, col] = find(cells);
end

function value = isnonempty(array)
    value = ~isempty(array);
end
function [SpringStrings, p, s] = PlotFunc(grElemNodeNrs, ...
    nodeNrCoords, cData, UpDir, MeaNodes, cFaces, ...
    StepThroughGroups, Eformat)

% INPUTS
% grElemNodeNrs = 2D cell array. grElemNodeNrs{M,N} is either
% 1) a row vector containing the node numbers for
% the Nth element of the Mth group, or
% 2) a K x 2 array with node numbers for springs in the
% first row. The corresponding entry in the second row
% is an integer (1−6) telling in which degree of
% freedom that the spring is acting.
% # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
% grElemNodeNrs = 2D cell array. grElemNodeNrs{M,N} is either
% 1) a column vector containing the node numbers for
% the Nth element of the Mth group, or
% 2) a 2-column array with node numbers for springs
% in the first column. The corresponding entry
% in the second column is an integer (1−6)
% telling in which degree of freedom that the
% spring is acting.
% nodeNrCoords = a four column matrix with the first column listing all
% node numbers. The remaining three columns list the
% coordinates of each node.
% cData = a column vector with cData(M) being a number used for
% the colour scaling when plotting the Mth group.
% UpDir = 1, 2 or 3. Tells which coordinate axis that shall be
% directed upwards when plotting
% MeaNodes = a possibly empty vector containing node numbers for
% nodes that shall be highlighted with a red star in the
% plot.
% cFaces = optional string with one of the following values
% 'none': No coloring of faces
% 'transparent': Transparent faces coloured
% (default value)
% StepThroughGroups = true or false. OPTIONAL. Default value: false.
% If true then one group of elements is highlighted
% at the time. Hit any key on the keyboard to show
% the next group.
% Eformat = optional parameter (still in experimental stage)
% for how the values of updating parameters should
% be viewed or not.
% Current possible values: 'FloatPt', 'GPa' or 'None')
% OUTPUT
% SpringStrings = Cell array of strings summarizing node point and cData
% value for each spring. Each spring is also marked with a
% red S in the plot

import UtilityPkg.*
%import UtilityPkg.GeneralUtilities %????????

[nRows, nCols] = size(grElemNodeNrs);
for rr = 1:nRows
    for cc = 1:nCols
        grElemNodeNrs{rr, cc} = grElemNodeNrs{rr, cc}';
    end
end

% # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
% % = = = Check inputs and set default values = = = = = = = = = = = = = = = = = = = = = = = = = =
if nargin < 8
    Eformat = 'None';
eend
if nargin < 7
    StepThroughGroups = false;
eend
TransparentFaces = false;
if nargin < 6
    cFaces = 'transparent';
eend
elseif strcmp (cFaces, 'flat')
    FaceColor = 'flat';
elseif strcmp (cFaces, 'none')
    FaceColor = 'none';
else
    error (['Unknown value ' ' cFaces ' ' for input string cFaces.'])
eend
% % Check if the modeled structure is all in the xy-, yz- or xz-plane
% This can of course be changed later to detect ANY plane, by using
% some elementary linear algebra.
PlanarStruct = true;
if all (nodeNrCoords(:,2)==nodeNrCoords(1,2))
    % 2D model in the yz-plane
    UpDir = 1;
elseif all (nodeNrCoords(:,3)==nodeNrCoords(1,3))
    % 2D model in the xz-plane
    UpDir = 2;
elseif all (nodeNrCoords(:,4)==nodeNrCoords(1,4))
    % 2D model in the xy-plane
    UpDir = 3;
else
    PlanarStruct = false;
eend
if ~(UpDir==1 || UpDir==2 || UpDir==3)
    error ('The input parameter UpDir must have the value 1, 2 or 3.')
eend
% Shift coordinate columns so that the last one is the "up direction"
cirshift ( , , 2) for shifting columns
cirshift (nodeNrCoords(:,2:4),'-',-UpDir), ';

[nofGroups, maxNofElements] = size (grElemNodeNrs);
if PlanarStruct
    % Join all shell elements i one group to one big polygon
    % quick & dirty...
    groupElemNodeCoords = cell (nofGroups, maxNofElements);
    groupElemNodeCoords = UtilityPkg.GeneralUtilities.polyg2Clockwise(grElemNodeNrs,nodeNrCoords,false);
    groupElemNodeCoordsUnion = cell (size (grElemNodeNrs));
    maxNofElements = 0;
    for gg = 1:ofGroups
        polygCoordCell = UtilityPkg.GeneralUtilities.polygUnion (groupElemNodeCoords(gg,:));
        groupElemNodeCoordsUnion (gg) = ...
        UtilityPkg GeneralUtilities.nodeCoords2NodeNrs (... polygCoordCell , nodeNrCoords);
        tmp = UtilityPkg.GeneralUtilities.polygUnion ( ...
            grElemNodeNrs (gg,:), nodeNrCoords);
```plaintext
tmpLen=length(tmp);
maxNofElements=max(maxNofElements,tmpLen);
groupElemNodeCoordsUnion(gg,1:tmpLen)=tmp;
end
groupElemNodeCoordsUnion=groupElemNodeCoordsUnion(:,1:maxNofElements);
grElemNodeNrs=groupElemNodeCoordsUnion;
[nofGroups,maxNofElements]=size(grElemNodeNrs);
end

%groupElemNodeCoords

% = = = Initial values = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

SpringStrings={};

%clf % Clear figure

if strcmp(Eformat,'FloatPt')
    % Do nothing
elseif strcmp(Eformat,'GPa')
cData=round(cData./1E9)*10;
else if strcmp(Eformat,'None')
    % Do nothing
end

% XYZcoord=circshift([1:2:3],-UpDir,2); % Pick any z-value
% ConstantZcoord=true; % Start value

%grElemNodeNrs = GroupBoundaries2D(grElemNodeNrs, false)

% First count the number of needed patches, the number of % springs and the max number of vertices
MaxNofVertices=0;
nofPatches=0;
nofSprings=0;
for GroupNr=1:nofGroups
    for ElemNr=1:maxNofElements
        [nRows,nCols]=size(grElemNodeNrs{GroupNr,ElemNr});
        if nCols==1 % If not spring(s)
            MaxNofVertices=max(MaxNofVertices,nRows);
            nofPatches=nofPatches+1;
            nofSprings=nofSprings+1;
        end
    end
end
patchNr=0;

% = = = Reserve memory for patches and springs = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
X=NaN(MaxNofVertices,nofPatches);
Y=NaN(MaxNofVertices,nofPatches);
Z=NaN(MaxNofVertices,nofPatches);
if nofSprings>0
    SPRINGS=true;
xSprings=zeros(nofSprings,1);
ySprings=zeros(nofSprings,1);
zSprings=zeros(nofSprings,1);
else
    SPRINGS=false;
end

if ischar(cData)
    if length(cData)>1
        C=zeros(nofPatches,3);
        RGBcolors=true;
    else
        oneColour=true;
        RGBcolors=false;
    end
else
    C=zeros(nofPatches,1);
end
```

---

This code snippet is likely part of a larger program that deals with handling arrays and performing various mathematical operations. The specific purpose isn't clear from the code alone, but it seems to involve arrays of varying lengths, possibly related to nodes and groups within a structure. The code includes error handling for string comparisons, array manipulations, and checking the length of data vectors.
RGBcolors=false;
oColour=false;
end

springCtr=0;
for GroupNr=1:nofGroups
    ElemNr=1;
xSum=0;
ySum=0;
zSum=0;
TotNofPts=0;
SprElem=false;
end

CurrentElem=grElemNodeNrs(GroupNr,ElemNr);
[nofNodes,nofCol]=size(CurrentElem);
nodeNrs=CurrentElem(:,1);
xCoords,yCoords,zCoords= ... UtilityPkg.GeneralUtilities.nodeNr2Coords(nodeNrs,nodeNrCoords);

if nofCol==2
    % Spring element
    SpringElem=true;
    DGFnrs=CurrentElem(:,2);
    nofSprings=nofNodes;
    lastNodeNr=NaN;
    for nn=1:nofSprings
        SpringStrings=[SpringStrings
                    ['Spring constant value at node ' int2str(nodeNrs(nn)) ...
                    ', DOF ' int2str(DGFnrs(nn)) ', ' num2str(cData(GroupNr))]]; %
        if lastNodeNr=nodeNrs(nn)
            % If the previous spring was not positioned
            % at the same node number, then save the
            % coordinates for plotting.
            springCtr=springCtr+1;
            xSprings(springCtr)=xCoords(nn);
            ySprings(springCtr)=yCoords(nn);
            zSprings(springCtr)=zCoords(nn);
        end
    end
else if (nofCol==1) && (nofNodes>1)
    % Beam or shell element
    patchNr=patchNr+1;
    if RGBcolors
        C(patchNr,:)=UtilityPkg.GeneralUtilities.ColorSpecStr2RGB(cData(GroupNr));
    else
        C(patchNr,:)=cData(GroupNr);
    end
    ind=[1:nVert ones(1,(MaxNofVertices−nVert))];
    X(:,patchNr)=xCoords(ind);
    Y(:,patchNr)=yCoords(ind);
    Z(:,patchNr)=zCoords(ind);
else
    error('One column for beam/shell element, two columns for springs.');
end
ElemNr=ElemNr+1;

if ~SpringElem
    NofElements=ElemNr−1;
    if ~StepThroughGroups
        CenterPtXYZ=[xSum/TotNofPts ySum/TotNofPts zSum/TotNofPts ];
    if strcmp(Eformat,'FloatPt')
        Estr = sprintf('%1.2g',cData(GroupNr));
    elseif strcmp(Eformat,'GPa')
        Estr = num2str(round(cData(GroupNr)/1E9+10)./10);
    elseif strcmp(Eformat,'None')
        Estr='';
    end
end
% text(CenterPtXYZ(1),CenterPtXYZ(2),CenterPtXYZ(3),Estr,...
% 'HorizontalAlignment','center','FontSize',30)
hold all
text(CenterPtXYZ(1),CenterPtXYZ(2),CenterPtXYZ(3),Estr,...
'HorizontalAlignment','center')
end
end
end

%% Plot marker for strings and measurement points

hold all
if springCtr˜=0
    xSprings=xSprings(1:springCtr);
ySprings=ySprings(1:springCtr);
zSprings=zSprings(1:springCtr);
else
    xSprings=[];
ySprings=[];
zSprings=[];
end
SpringColorRGB=[1 0 0]; % red

% Mark measurement nodes (if any) with red stars
MeaPtColorRGB=[0 0 1]; % blue
nofMeaNodes=length(MeaNodes);
if nofMeaNodes>0
    [xMeaPts,yMeaPts,zMeaPts]...=UtilityPkg.GeneralUtilities.nodeNr2Coords(MeaNodes,...
    nodeNrCoords);
else
    xMeaPts=[];
yMeaPts=[];
zMeaPts=[];
end
if springCtr˜=0 || nofMeaNodes = 0
    xx=xSprings;xMeaPts;
yy=ySprings;yMeaPts;
zz=zSprings;zMeaPts;
    MarkerArea=18;
    MarkerColorRGB=[repmat(SpringColorRGB,springCtr ,1)
    repmat(MeaPtColorRGB,nofMeaNodes,1)];
    s=scatter3(xx,yy,zz,MarkerArea,MarkerColorRGB,'MarkerFaceColor','flat');
else
    s=NaN;
end
%
if oneColour
    p=patch(X,Y,Z,cData);
    set(p,'EdgeColor',cData,'LineWidth',2);
else
    RGBcolors
    %If
    % From http://www.mathworks.com/matlabcentral/newsreader/view_thread/276684 :
    % "If you are using transparency, then you must be using the OpenGL renderer."
    %
    % See also: http://abarry.org/speed-up-matlab-figures-with-opengl/
    set(gcf,'Renderer','OpenGL') % Just in case OpenGL is not default. (CHECK LATER!!!)
    p=patch(X,Y,Z,C);
    set(p,'FaceColor','flat','FaceVertexCData',C,'LineWidth',2)
else
    %MeaPtSize=100;
    %MeaPtColor='r';
    %Cedges=repmat(C',MaxNofVertices,1);
    if MaxNofVertices==2
        % Lines only
        p=patch(X,Y,Z,[C C]','EdgeColor','flat');
    else
        % TO DO:
        % * If both lines and 2D elements, fix colouring of lines!
% Probably exactly like for lines only above.
% * Join patches with common edges and normal vectors to larger patches before plotting.

%% Really stupid bug fix
%% In the following patch command, the colour must be given as a COLUMN vector, or patch will generate an error message in Matlab R2012 (but not in R2014 or newer).
%% But then, if there are three patches to plot and thus C has length 3, then patch will misunderstand C to be an RGB colour triplet and generate an error if not all elements are in the range [0,1].
%% Thus, to avoid this bug, we print the third patch twice in such cases.
if length(C)==3
    C=[C;C(end)];
end
X=X(:,1:end end);
Y=Y(:,1:end end);
Z=Z(:,1:end end);
end
p=patch(X,Y,Z,C,'FaceColor','flat','LineWidth',2);
end
end
if TransparentFaces
    set(p,'FaceAlpha',0.5);
end

%% Plotting details
if PlanarStruct
    view([0 90]) % view from top
else
    view([135 45]) % For non-flat structures
end
xlabel('x');
ylabel('y');
zlabel('z');
axis tight
axis equal
set(gca,'FontSize',32) % For paper on the plate
colormap(jet)
colorbar

function RGB=ColorSpecStr2RGB(str)
    if strcmp(str,'y')
        RGB=[1 1 0];
    elseif strcmp(str,'m')
        RGB=[1 0 1];
    elseif strcmp(str,'c')
        RGB=[0 1 1];
    elseif strcmp(str,'r')
        RGB=[1 0 0];
    elseif strcmp(str,'g')
        RGB=[0 1 0];
    elseif strcmp(str,'b')
        RGB=[0 0 1];
    elseif strcmp(str,'w')
        RGB=[1 1 1];
    elseif strcmp(str,'k')
        RGB=[0 0 0];
    end
end

function [xCoords,yCoords,zCoords]=nodeNr2Coords(nodeNrs,nodeNrCoords)
    % nodeNrs =
    % nodeNrCoords=
    nodeNrs=nodeNrs(:); % Make column vector
    nofNodes=length(nodeNrs);
    xCoords=zeros(nofNodes,1);
yCoords=zeros(nofNodes,1);
zCoords=zeros(nofNodes,1);

for nn=1:nofNodes
    ind=find(nodeNrCoords(:,1)==nodeNrs(nn));
    if length(ind)==1
        error('Unknown node number')
    end
    xCoords(nn)=nodeNrCoords(ind,2);
yCoords(nn)=nodeNrCoords(ind,3);
zCoords(nn)=nodeNrCoords(ind,4);
end

function GroupBoundaries = GroupBoundaries2D(GroupElemNodesCoords,DEBUG)
    [nofGroups, maxNofElements] = size(GroupElemNodesCoords);
    GroupBoundaries = cell(nofGroups,1);
    colormap(jet)
    for GroupNr=1:nofGroups
        GroupPolyX = [];
        GroupPolyY = [];
        for j = 1:maxNofElements
            if ~isempty(GroupElemNodesCoords(GroupNr,j))
                XY = GroupElemNodesCoords(GroupNr,j)(:,1:2);
                PolyX.PolyY = poly2cw(XY(:,1).XY(:,2));
                GroupPolyX, GroupPolyY = polybool('union', ...)
                GroupPolyX, GroupPolyY, PolyX, PolyY);
            end
        end
    end
    if DEBUG
        figure(21)
        [f, v] = poly2fv(GroupPolyX, GroupPolyY);
        patch('Faces', f, 'Vertices', v, 'FaceColor', 'flat', ...)
        'FaceVertexCData',GroupNr, 'EdgeColor', 'none')
        hold on
    end
    GroupBoundaries{GroupNr,1} = [GroupPolyX, GroupPolyY];
end

%%%%

%% Computes the following subindices:
%% Input:
%% TestNodeVector — node numbers for the original test mode shape in the
%% order corresponding to the data in this test mode shape
%% Test2FemPlotNodeMap — N×2, map between test and FEM plotted node numbers in any
%% order with only restriction that the 1st column
%% corresponds to the plotted test node numbers and 2nd column
%% corresponds to the plotted FEM node numbers
%% TestMeaDofs — M×4, map between test node numbers and measured dofs.
%% The measured dofs for each node is 1×3 array, with zeros
%% and ones, where 1 at position n in {1 (=test x axis),2
%% (=test y axis), 3 (=test z axis)} in this array
%% means that the measurement was done along n axis of the
%% test coordinate system. 1st column in this map
%% corresponds to test node numbers in any order.
%% Test2FemXYZ — 3×1 mapping between test model xyz coordinate system
%% and FE model xyz coordinates. For example, in Abaqus
%% the FEM coordinate system is taken as the global
%% coordinate system which is right-handed and orthogonal.
%% Negative component in this mapping means that given
%% test coordinate axis is in opposite direction to the
%% corresponding FE model coordinate axis. We also assume
%% that the given mapping defines a right-handed orthogonal
%% coordinate system for the test model, otherwise an error
%% will be generated. In general, the test model and the
%% FE model coordinate systems are well matched to each
%% other since the FE model uses one of the axes pointing
%% along the gravity line and one along the side of the structure,
%% for example, along the bridge deck. At the same time
%% the gravity direction is also a natural choice in the
%% test data in the sense of simple adjustment of the
%% accelerometer to the structure as well as to the direction
% of some side of the structure.
% NrOfDofsPerTestNode — number of degrees of freedom per test node.
% This number is assumed to be the same for each test node.
% NrOfDofsPerFemNode — number of degrees of freedom per FEM node. This number is assumed to be the same for each FEM node.
% Output:
% 1. phiTestPlotInd — subindices in the test mode shape for the plotted dimensions (nodes + dofs)
% 2. phiFemPlotInd — subindices in the FEM mode shape for the plotted dimensions (nodes + dofs) that correspond to phiTestPlotInd.
% phiTestPlotInd and phiFemPlotInd have the same length and the data in test and FEM mode shapes which correspond to these indices is the correlated data.
% The order of the data that correspond to these indices is the same as order of test nodes in the Test2FemPlotNodeMap and the order of degrees of freedom for each node corresponds to x, y and z FEM coordinate axis.
% 3. phiMeaFemInd — subindices in the FEM mode shape corresponding to the measured dimensions (nodes + dofs).
% 4. phiPlotSubInd — subindices in the phiTestPlotInd and therefore in the phiFemPlotInd that correspond to the measured dimensions (nodes + dofs).
% The data that corresponds to phiMeaFemInd indices in FEM mode shape.
% and phiPlotSubInd in phiFemPlotInd in the FEM mode shape is exactly the same data.
% Note! The sequence of nodes corresponds to the sequence of nodes from the Artemis mode shape file and the dofs are in the order of FEM coordinate system.

function [phiTestPlotInd, phiFemPlotInd, phiResidInd, residSubInd] = ...
    MakeIndexVectors(testNodeVector, test2FemPlotNodeMap, testResidDofs, ...
    test2FemXYZ, ...
    nrOfDofsPerTestNode, nrOfDofsPerFemNode)

import UtilityPkg.*

Fem2TestXYZ = GeneralUtilities.revCoordTrans(test2FemXYZ);
% TestMeaDofs(:,1) <= Test2FemNodes(:,1) <= TestNode
if ~all(ismember(testResidDofs(:,1),test2FemPlotNodeMap(:,1)))
    error(‘Measurement nodes are not a subset of a test model!’);
end
if ~all(ismember(test2FemPlotNodeMap(:,1),testNodeVector))
    error(‘Nodes in a test model is not a subset of a test data nodes!’);
end

% Rearrange input data so this is in increasing order of FEM mode shape number, thus computed PhiTestPlotInd, PhiPlotSubInd and PhiMeaFemInd will be vectors with increasing components.
% Redefine Test2FemPlotNodeMap so 2nd column is in increasing order
[~, ind] = sort(test2FemPlotNodeMap(:,2),’ascend’);
test2FemPlotNodeMap = test2FemPlotNodeMap(ind,:);
% Redefine TestMeaDofs so 1st column is in increasing order of corresponding FEM components
newTestMeaDofs = testResidDofs;
% remove all rows with 2:end components equal to zero
newTestMeaDofs(all(newTestMeaDofs(:,2:end)==0,2),:) = [];
[v, ind] = ismember(newTestMeaDofs(:,1),test2FemPlotNodeMap(:,1));
subInd = find(v);
M = [test2FemPlotNodeMap(ind(subInd),:) newTestMeaDofs(subInd,2:end)];
[~, ind] = sort(M(:,2),’ascend’);
M = M(ind,:);
newTestMeaDofs = M(:,[1 3 4 5]);

% Define FemMeaDofs so 1st column is increasing order of FEM node number. Note, it also corresponds to NewTestMeaDofs
M(:,3:end) = M(:,abs(Fem2TestXYZ)+2);
newResidDofs = M(:,2:end);
% Find PhiFemPlotInd
% node order is according to Test2FemPlotNodeMap 2nd column and therefore in increasing order of components
newNodes = test2FemPlotNodeMap(:,2);
base = (femNodes-1)*nrOfDofsPerFemNode;
\texttt{phiFemPlotInd} = \texttt{phiFemPlotInd}();

\% Find \texttt{phiPlotSubInd}
\% node order is according to updated NewTestMeaDofs 1st column and
\% therefore in increasing order of FEM components.
\[v, MeaInd\] = ismember(newTestMeaDofs(:,1), test2FemPlotNodeMap(:,1));
subMeaInd = find(v);
residSubInd = [ ];
for i=1:length(subMeaInd)
    TestBase = (MeaInd(subMeaInd(i))-1) \times nrOfDofsPerTestNode;
    TestDofs = newTestMeaDofs(subMeaInd(i), 2:end);
    offset = sort(abs(test2FemXYZ(TestDofs \neq 0)).'ascend');
    residSubInd = [ residSubInd TestBase+offset ];
end
residSubInd = residSubInd();;

\% Find \texttt{phiFemResidInd}
\% node order is according to updated FemMeaDofs 1st column and
\% therefore in increasing order of FEM components.
phiFemResidInd = [ ];
for i=1:length(femResidDofs)
    base = (femResidDofs(i,1)-1) \times nrOfDofsPerFemNode;
    offset = find(femResidDofs(i,2:end));
    phiFemResidInd = [ phiFemResidInd base+offset ];
end
phiFemResidInd = phiFemResidInd();;

\% Convert transformation vector \texttt{xyz1TOxyz2} which defines \texttt{xyz1}
\% coordinate axes in terms of axes from right-handed coordinate
\% system \texttt{xyz2} to a reverse transformation vector \texttt{xyz2TOxyz1}.
\% This function is used, for example to convert Test2FemXYZ to
\% Fem2TestXYZ transformation vectors.
\% Input:
\% \texttt{xyz1TOxyz2} - 3x1 or 1x3 transformation vector which defines \texttt{xyz1}
\% coordinate axes in terms of \texttt{xyz2} coordinate axes. For example,
\% let \texttt{xyz1TOxyz2}=\texttt{[a b c]}, where \texttt{abs(a), abs(b), abs(c)} are 3 different
\% numbers from the set \{1, 2, 3\}. Note, \texttt{a, b, c} can be negative. Then,
\% \texttt{x} axis of \texttt{xyz1} coordinate system corresponds to a axis of
\% \texttt{xyz2} coordinate system, \texttt{y} axis of \texttt{xyz1} coordinate system corresponds
\% to \texttt{b} axis of \texttt{xyz2} coordinate system and \texttt{z} axis of \texttt{xyz1} coordinate
\% system corresponds to \texttt{c} axis of \texttt{xyz2} coordinate system.
\% Output:
\% \texttt{xyz2TOxyz1} - the reverse transformation which defines \texttt{xyz2} coordinate
\% system axes in terms of \texttt{xyz1} coordinate system axes.
\% System \texttt{xyz2TOxyz1} is a right-handed coordinate system.
\% The function is a left-handed version of \texttt{xyz1TOxyz2}.
\% \texttt{xyz2TOxyz1} = \texttt{revCoordTrans(xyz1TOxyz2)}
\% Syntax: \texttt{xyz2TOxyz1 = \texttt{revCoordTrans(xyz1TOxyz2)}}
\% error ( 'The coordinate system defined by transformation ' + ...
\% numstr(xyz1TOxyz2)+'' from \{1, 2, 3\} is not a right-handed!')
end
v = find(xyz1TOxyz2<0);
t = abs(xyz1TOxyz2);
toX = find(t==1);
toY = find(t==2);
toZ = find(t==3);
if ismember(toX,v)
toX = -toX;
end
end

if ismember(toY,v)
    toY = -toY;
end

if ismember(toZ,v)
    toZ = -toZ;
end

xyz2TOxyz1 = [toX toY toZ]';
end

% Check if the coordinate system given by its axis transformations
% to a right-handed coordinate system is right-handed itself.
%
% Rule: Let xyz1TOxyz2 be a transformation vector from xyz1 to xyz2
% coordinate system. See explanation below. Then, if the sum of
% the number of swaps of needed to transform abs(xyz1TOxyz2) to [1 2 3]
% and the number of negative elements in xyz1TOxyz2 is even, then
% the xyz1 coordinate system is right-handed, otherwise it is not.
%
% Let xyz1TOxyz2=[a b c], where abs(a), abs(b), abs(c) are 3 different
% numbers from the set {1,2,3}. Note, a,b,c can be negative. Then,
% x axis of xyz1 coordinate system corresponds to a axis of
% xyz2 coordinate system, y axis of xyz1 coordinate system corresponds
% to b axis of xyz2 coordinate system and z axis of xyz1 coordinate
% system corresponds to c axis of xyz2 coordinate system.
%
% Input:
% xyz1TOxyz2 – the transformation vector which defines the axes of
% xyz1 coordinate system in terms of axes of the right-handed
% coordinate system xyz2.
% Output:
% true if the xyz1 coordinate system is also right-handed, false otherwise

function isRightHanded = isRightHanded(xyz1TOxyz2)
    t = abs(xyz1TOxyz2);
    if ~ all([ find(t==1) find(t==2) find(t==3)])
        error('Some of the axes are missing')
    end
    v = find(xyz1TOxyz2<0);
    ng = length(v);
    nrOfSwaps = 0;
    i1 = find(t==1);
    if i1>1
        tmp = t(1);
        t(1) = 1;
        t(i1) = tmp;
        nrOfSwaps = nrOfSwaps + 1;
    end
    i2 = find(t==2);
    if i2>1
        tmp = t(2);
        t(2) = 2;
        t(i2) = tmp;
        nrOfSwaps = nrOfSwaps + 1;
    end
    if (t(1)==1 || t(2)==2 || t(3)==3)
        error('Internal error, should never happen');
    end
    isRightHanded = ~mod(ng + nrOfSwaps,2);
end

% Read ascii file where the updating groups are defined.
% This text file is used for more easy and flexible definition of
% the updating groups. It should follows the following
% requirements:
% 1. Different groups in this file shall be separated by at least one blank line.
% 2. Blank line may contain zero or any number of whitespaces.
% 3. Line which starts with # (comment delimiter) is a comment line.
% 4. The comment line continues until the end of the line.
% 5. It is ignored when the file is processed. Comment line is not
% considered as a group separation line.
% 6. Every spring group shall precede with the tag sPRING in a separate line
% before the corresponding group. It is allowed to have any number of blank lines
```matlab
% after the spring group tag and before the beginning of the group definition.
function [GrNr2GrNameMap, springGrIndArray, notSpringGrIndArray, ...
    nrOfGroups, nrOfSpringGroups] = ReadGroupDefFile(GroupDefFilName)
    SPRING = 'SPRING';
    SPRINGLEN = length(SPRING);
    GrNr2GrNameMap = cell(1,1);
    springGrIndArray = [];
    ReadGrNr = 1;
    NrOfElemsInGroup = 0;
    FillNewGroup = true;
    fid = fopen(GroupDefFilName);
tline = fgets(fid);
while ischar(tline) % if tline is array of characters
    tline = strtrim(tline);
    if ~isempty(tline)
        if tline(1) == '#'
            % not a comment line
            if FillNewGroup
                disp(['Group #', num2str(ReadGrNr)])
            end
        end
        if length(tline) >= SPRINGLEN && strcmp(tline(1:SPRINGLEN), SPRING)
            if isempty(GrNr2GrNameMap{1})
                springGrIndArray = [springGrIndArray 1];
            else
                springGrIndArray = [springGrIndArray size(GrNr2GrNameMap,1)+1];
            end
        end
    else
        NrOfElemsInGroup = NrOfElemsInGroup + 1;
        GrNr2GrNameMap{ReadGrNr,NrOfElemsInGroup} = tline;
        FillNewGroup = false;
    end
    disp(tline)
end
else
    if ~FillNewGroup
        FillNewGroup = true;
        ReadGrNr = ReadGrNr + 1;
        NrOfElemsInGroup = 0;
    end
    disp(tline)
end
fclose(fgetl(fid));
end
nrOfSpringGroups = length(springGrIndArray);
rnOfGroups = size(GrNr2GrNameMap,1);
tmp = zeros(nrOfGroups,1);
tmp(springGrIndArray) = ones(nrOfSpringGroups,1);
notSpringGrIndArray = find(tmp==0);
end
function Topology = createGridTopology(NrOfRows, NrOfCols)
    Topology = cell(1,1);
    NrOfElems = 0;
    N=NrOfRows*NrOfCols;
    k=1;
    while k<=N-NrOfRows+1
        if mod(k,NrOfRows)+1==0
            Topology{1,NrOfElems+1} = [k k+NrOfRows];
        elseif mod(k,NrOfRows)==0
            Topology{1,NrOfElems+1} = [k k+NrOfRows];
        end
        NrOfElems = NrOfElems + 1;
    end
end
```

end
    k = k+1;
end
while k<=N
    Topology{1,NrOfElems+1} = [k k+1];
    NrOfElems = NrOfElems + 1;
    k = k+1;
end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

NDRA INDATA SAMT KR MED
% xCoords, yCoords, zCoords = ...
% UtilityPkg_GeneralUtilities_nodeNr2Coords(nodeNrs, nodeNrCoords);
% FR KOORDINATER!!!!

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function xyClockwise = polyg2Clockwise(grElemNodeNrs, nodeNrCoords, checkInters)

% USAGE: xyClockwise = polyg2Clockwise(xy, checkInters)
% DESCRIPTION: Function for arranging the vertices of simple polygons in
clockwise order. It also removes unnecessary repeated vertices.
% It also works for a certain type of self-intersecting polygons,
as described below.
% INPUTS
% xy = two-column matrix or a cell array of two-column matrices.
% The rows in each such matrix are the coordinates of the
% vertices of a simple polygon.
% It is allowed but not required that xy(end,:) = xy(1,:).
% Repetitions xy(m,:) = xy(m+1,:) are also allowed.
% Repetitions of both these kinds are removed in the output.
% checkInters = true or false (optional with default value false)
% If true, then an error message will be generated if xy
% contains a self-intersecting polygon. Checking this
% is more time consuming.
% OUTPUTS
% xyClockwise = two-column array or cell array the same size as xy with
% each entry being a copy of the corresponding entry in xy, but with the rows order flipped if necessary to give
% clockwise ordering of the polygon vertices.
% Polygons with <= 2 unique vertices are considered being clockwise.
% If checkInters=false, then clockwise direction is correctly
% detected also for polygons with linear "spurs" like
% xy'=[0 1 1 0 0 0 1 0]
% 0 0 1 1 0 0 1 0]
% For other types of intersections, the polygon is considered
% clockwise if the signed area is positive, as defined on
% http://geomalgorithms.com/a01_area.html
% (This command is very similar to the Matlab Mapping Toolbox command
% poly2cw.)
% EXAMPLE USAGE:
% t=linspace(0,5.5*pi,100),';
% x=[t.*cos(t); -6*pi; 0];
% y=[t.*sin(t); -6*pi; 0];
% plot(x,y,'b,-');
% ispolycw(x,y)
% polyglsClockwise([x y])
% polyglsClockwise([x y],true) % Will generate error message
if nargin<3
function a = SignedArea(xy)
% a = SignedArea(xy) returns the signed area of the simple polygon with

end
% vertex coordinates saved in the rows of xy.
% Inputs
% xy = two-column matrix with the rows defining the vertices of a polygon.
% Outputs
% a = the signed area of the polygon as described in
% http://geometryalgorithms.com/Archive/algorithm_0101/algorithm_0101.htm
% Example
% xy=[0 1 1 0 ; 0 0 1 1]'; % A square, counter clockwise, area 1.
% a=SignedArea(xy)
% xy=[0 -1 0 1 ; -1 0 1 0]'; % A square, clockwise, area 2.
% a=SignedArea(xy)
N = numel(xy(:,1));
if N <= 2
    a = 0;
else
    nMinusOne = [N 1:N-1];
    nPlusOne = [2:N 1];
    n = (1:N);
    a = sum(xy(n,1).* (xy(nPlusOne,2)-xy(nMinusOne,2)))/2;
end

function tf=lineSegmentsIntersect(A,B,C,D)
% Usage: tf=lineSegmentsIntersect(A,B,C,D)
% Description: Tells whether the line segment from A to B intersects with
% the line segment from C to D.
% Inputs
% A,B,C,D = length two column vectors giving the coordinates of four points
% in the plane.
% Output
% tf = true if the line segment from A to B intersects with the line
% segment from C to D.
% false otherwise
% Note that the line segments A-B and C-D intersect if and only if the
% following two conditions both hold true
% 1) A and B are both on different sides of the line through C and D,
% OR at least one of A and B are on this line.
% 2) C and D are both on different sides of the line through A and B,
% OR at least one of C and D are on this line.
% We check 1) by computing signed areas that have the same sign if A and B
% are on the same side of the line and equals zero when one of the points
% is on the line.
% Parallel or collinear line segments
sA1=signedArea([C(1);D(1):A(1)];[C(2);D(2):A(2)]);
sA2=signedArea([C(1);D(1):B(1)];[C(2);D(2):B(2)]);
if ((sA1==0) && (sA2==0))
    AB=[B(1)-A(1);B(2)-A(2)];
    AC=[C(1)-A(1);C(2)-A(2)];
    AD=[D(1)-A(1);D(2)-A(2)];
    v=AB./norm(AB);
    c=v'*AC;
    d=v'*AD;
    if ( (c>=0 && c<=1) || (d>=0 && d<=1) )
        tf=true;
    else
        tf=false;
    end
else
    cond1 = sA1*sA2<=0;
    % Similarly for condition 2:
end
tf = cond1 && cond2;
end
end

function errIfSelfIntersect(xy)
% INPUT
% xy = two-column matrix with the rows defining the vertices of
% a polygon
% OUTPUT
% Generates an error message if the polygon contains any intersecting
% edges
nVert = numel(xy(:,1));
SelfIntersection = false;
xyExp = xy([1:end 1],:);
for mm = 1:(nVert-2)
% As described in the note in the documentation above, the line
% segment between vertices mm+2 and mm+3 is the next one to check.
    for nn = mm+2:nVert
        if lineSegmentsIntersect(xy(mm,:),xy(mm+1,:)',...,xy(nn,:),xyExp(nn+1,:))
            SelfIntersection = true;
        end
    end
end
if SelfIntersection
    error('Self-intersecting polygon!')
end
end

function ind = findLeftmostLowestVertices(xy)
% Return the indices of all the left-most lowest vertices in xy.
% Find the vertices with the minimum y.
[~, ind] = min(xy(:,2));
[~, subInd] = min(xy(ind,1));
ind = ind(subInd);
end

function v = removeRepElem(w)
% Removes repeated elements in the input vector w (cyclic indexing)
% INPUTS
% w = vector
% OUTPUTS
% v = vector obtained from v by removing every element w(n) such
% that w(n)==w(n+1) or w(end)==w(1)
wlen = numel(w);
if wlen>1
    ind = 1:wlen;
    indNext = [2:wlen 1];
    nonDuplInd = find(w(ind,:)~=w(indNext));
    v = w(nonDuplInd,:);
else
    v = w;
end
end

function polygUnionNodeNrs = polygUnion(polygVertNodeNrs, nodeNrCoords)
% DESCRIPTION: VERY simple function for computing the union of polygons
% PROVIDED that
% * They all have CLOCKWISE ordering of the vertices
% * All polygon intersections are full edges.
% * No intersection of interiors or of proper subsets of edges.
% * No polygon contains a hole
The union of the polygons contains no hole.
The union is a connected set with no holes.
Neither of these properties are checked by this function.

If the the input groups of elements contains a spring element
or a line that is not edge of a polygon, then the output generated
is polygUnionNodeNrs=polygVertNodeNrs
Without these restrictions, a more complicated algorithm is needed
(provided, for example, via the ploybool command in the Matlab
Mapping Toolbox).

USAGE: polygVertCoordsU = polygUnion(polygVertCoords)

INPUTS
polygVertCoords = cell array of two-column matrices.
The rows in each such matrix are the coordinates of the
vertices of a simple polygon.
It is NOT allowed that polygVertCoords(end,:)=polygVertCoords(1,:).
No repetitions polygVertCoords(m,:)=polygVertCoords(m+1,:) are allowed.

OUTPUTS
polygUnionNodeNrs = cell array containing one or more column vectors

DEBUG = true;

First reduce the number of rows in nodeNrCoords to nodes included
in polygVertNodeNrs
nodeNrCoords=CompactNodeNrCoords(polygVertNodeNrs,nodeNrCoords);

% Make a bookkeeping matrix edgeFromTo such that NDRA !!!!!
% edgeFromTo(m,n)=true if there is at least one polygon edge
going from vertex number m to vertex number n.
% =false otherwise.
% =NaN if there is a line segment but no polygon edge
% and one bookkeeping matrix edgeInLineButNotInPolyg such that
% edgeInLineButNotInPolyg(m,n)=true if there is a line segment (=beam
% element) but no polygon edge going
% from vertex number m to vertex number
% =false otherwise.
% [nGroups,maxNoFElemInGroup]=size(polygVertNodeNrs);
% [totNoFVert] = size (nodeNrCoords); % Number of unique vertices
edgeFromTo=false(totNoFVert);
edgeInLineButNotInPolyg=false(totNoFVert);
noSprings=true;

while noSprings && gg<=nGroups
    [~,nCols]=size(polygVertNodeNrs{gg,ee});
    if nCols==2
        noSprings=false;
    else
        while ee<=maxNoFElemInGroup && isempty(polygVertNodeNrs{gg,ee})
            % Polygon node numbers expanded so that expPolyg(end)=expPolyg(1)
            expPolyg=polygVertNodeNrs{gg,ee}((1:end-1));
            % Indices such that nodeNrCoords(expPolygInd,1)=expPolyg
            expPolygInd=nodeNr2Ind(expPolyg,nodeNrCoords);
            if DEBUG && any(nodeNrCoords(expPolygInd,1)==expPolyg)
                error('Incorrectly constructed expPolygInd')
            end
            if length(expPolyg)==3
                % Line segment.
                if edgeFromTo(expPolygInd(1),expPolygInd(2))==false
                    % Line segment not yet found to be part of a polygon edge
                    edgeInLineButNotInPolyg(expPolygInd(1),expPolygInd(2))=true;
                end
            end
        end
    end
end
else
    for nn=1:length(expPolyg)-1
        edgeFromTo(expPolygInd(nn),expPolygInd(nn+1))=true;
        edgeInLineButNotInPolyg(expPolygInd(nn),expPolygInd(nn+1))=false;
    end
    ee=ee+1;
    end
end
if noSprings || any(edgeInLineButNotInPolyg(:))
    polygUnionNodeNrs=polygVertNodeNrs;
else
    polygUnionNodeNrs=findConnectedUnion(edgeFromTo,nodeNrCoords);
end

function nodeNrCoordsCompact...
    =CompactNodeNrCoords(polygVertNodeNrs,nodeNrCoords)
end

function nodeNrsInd=nodeNr2Ind(nodeNrs,nodeNrCoords)
    nodeNrsInd=zeros(size(nodeNrs));
    nNodeNrs=numel(nodeNrs);
    for nn=1:nNodeNrs
        tmp=find(nodeNrs(nn)==nodeNrCoords(:,1));
        if numel(tmp)>1
            error('Ohdearohdearohdear...')
        end
        nodeNrsInd(nn)=tmp;
    end
end

function nodeNrsOutside=...
    pointsOutsidePolygon(polygonNodeNrs,nodeNrCoords)
    nofVert=length(polygonNodeNrs(:));
    ind=zeros(nofVert,1);
    allNodeNrs=nodeNrCoords(:,1);
    allX=nodeNrCoords(:,2);
    allY=nodeNrCoords(:,3);
    for nn=1:nofVert
        ind(nn)=find(allNodeNrs==polygonNodeNrs(nn));
    end
    polyX=nodeNrCoords(ind,2);
    polyY=nodeNrCoords(ind,3);
    polyX=nodeNr2Ind(polygonNodeNrs,nodeNrCoords),2);
    polyY=nodeNr2Ind(polygonNodeNrs,nodeNrCoords),3);
% THE SOURCE CODE

nodeNrsOutside=allNodeNrs(find(~inpolygon(allX,allY,polyX,polyY)));  
end

function polygUnionNodeNrs=findConnectedUnion(edgeFromTo,nodeNrCoords)

[totNofVert,~]=size(nodeNrCoords); % Number of unique vertices

% Start from the leftmost lowest vertex and walk along the edges
% always choosing the outermost one
polygUnionVertNr=zeros(totNofVert,1);

polygUnionNodeNrs=zeros(totNofVert,1);

[ind,subInd]=min(nodeNrCoords(:,2)) ;
if numel(subInd)>1
    error(’Two different nodes with the same coordinates’)
end

startNodeInd=ind(subInd);

polygUnionNodeNrs(1,:)=nodeNrCoords(startNodeInd,1);

nodesFoundCtr=1; % Current number of nodes in the output polygon
lastFoundNodeInd=startNodeInd;
polygUnionNotClosed=true; % Tells wheter the found polygon is closed

while polygUnionNotClosed
    % Find all edges starting from the node given by lastFoundNodeInd
    candInd=find(edgeFromTo(lastFoundNodeInd,:));
    nofCandidates=length(candInd);

    % Each edge given by one column vector:
    V=nodeNrCoords(candInd,2:3)-repmat(nodeNrCoords(lastFoundNodeInd,2:3)',1,nofCandidates);
    % Since all input and output polygons have clockwise ordering of the vertices, we can pick the outermost of the candidate edges by finding a column u of V for which the cross product u x v with any other column vector of V gives a negative z-direction entry.
    while (length(V(1,:))>1)
        u=V(:,1);
        v=V(:,2);
        if u(1)*v(2)-u(2)*v(1)<0
            V=V(:,[1 3:end]);
        else
            V=V(:,2:end);
        end
    end

    lastFoundNodeInd=candInd;

    polygUnionNotClosed=(lastFoundNodeInd==startNodeInd);
    if polygUnionNotClosed
        nodesFoundCtr=nodesFoundCtr+1;
        polygUnionNodeNrs(nodesFoundCtr)=nodeNrCoords(lastFoundNodeInd,1);
    end

polygUnionNodeNrs=[polygUnionNodeNrs polygUnionNodeNrs];

% Now we have found the union of all polygons connected to the node
% we started from.
% Next check if there are any input polygons that
% are not connected to this union.
nodeNrsOutside=pointsOutsidePolygon(polygUnionNodeNrs,nodeNrCoords);
if ~isempty(nodeNrsOutside)
    ind=nodeNrs2Ind(nodeNrsOutside,nodeNrCoords);
    edgeFromTo=edgeFromTo(ind,ind);
    nodeNrCoords=nodeNrCoords(ind,:);
    polygUnionNodeNrs=[polygUnionNodeNrs ... findConnectedUnion(edgeFromTo,nodeNrCoords)];
else
    polygUnionNodeNrs=[polygUnionNodeNrs];
end
end
end
function groupElemNodeCoords ...

    = nodeNrs2NodeCoords(grElemNodeNrs,nodeNrCoords)

% INPUTS
% grElemNodeNrs, nodeNrCoords = same as for PlotFunc
% OUTPUT
% groupElemNodeCoords = cell array same size as grElemNodeNrs
% Each is either empty array or a matrix with
% either
% * three columns giving the coordinates of the
%   vertices of a simple polygon or a line
% * four columns for springs, with the first three
%   columns giving the coordinates and the fourth
%   one giving the DOF (= same as in
%  grElemNodeNrs)
% # # # # # # # # # element either
if iscell(grElemNodeNrs)
    groupElemNodeCoords = cell(size(grElemNodeNrs));
% noElem = numel(grElemNodeNrs);
[nRows, nCols] = size(grElemNodeNrs)
noElem = nRows * nCols;

ee = 1;
while ee <= noElem && ~isempty(grElemNodeNrs{ee})
    groupElemNodeCoords{ee} = ...
        = UtilityPkg.GeneralUtilities.nodeNrs2NodeCoords(grElemNodeNrs{ee},
            nodeNrCoords);
    ee = ee + 1;
end
else
    [nRows, nCols] = size(grElemNodeNrs);
    if nCols == 1
        % Beam or shell element
        groupElemNodeCoords = nodeNrs2Coord(grElemNodeNrs, nodeNrCoords);
    elseif nCols == 2
        % spring element
        groupElemNodeCoords = zeros(nRows, 4);
        groupElemNodeCoords(:, 4) = grElemNodeNrs(:, 2); % DOF column
        groupElemNodeCoords(:, 1:3) = ...
            = nodeNrs2Coord(grElemNodeNrs(:, 1), nodeNrCoords);
    else
        error(['Incorrect number of columns'])
    end
end

% Help function
function XYZ = nodeNrs2Coord(nodeNrs, nodeNrCoords)
% INPUTS
% nodeNrs = column vector
% nodeNrCoords=
% OUTPUTS
% XYZ = 3-column vector
nodeNrs = nodeNrs(:,);
NN = length(nodeNrs);
XYZ = zeros(NN, 3);
for nn = 1:NN
    noMatchFound = true;
    if isnan(nodeNrs(nn))
        XYZ(nn, :) = NaN NaN NaN;
    noMatchFound = false;
    elseif
        ind = find(nodeNrs(nn) == nodeNrCoords(:, 1));
        if length(ind) > 1
            error([ 'Repeated node number '])
        elseif ~isempty(ind)
            XYZ(nn, :) = nodeNrCoords(ind, 2:4);
        noMatchFound = false;
    end
end
if noMatchFound
error('Incorrect number of columns')
end
end
end
end

function grElemNodeNrs=nodeCoords2NodeNrs(groupElemNodeCoords, nodeNrCoords)
    % % = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    % % = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    function grElemNodeNrs=nodeCoords2NodeNrs(groupElemNodeCoords, nodeNrCoords)
        % INPUTS
        % groupElemNodeCoords= Same as in PlotFunc
        % nodeNrCoords = cell array same size as groupElemNodeCoords
        % Each is either empty array or a matrix with
        % either
        % * three columns giving the coordinates of the
        % vertices of a simple polygon or a line
        % * segment(=shell element or beam element) or
        % * four columns for springs, with the first three
        % columns giving the coordinates and the fourth
        % one giving the DOF (=same as in
        % grElemNodeNrs)
        %
        % OUTPUT
        % grElemNodeNrs = Same as in PlotFunc
        %
        if iscell(groupElemNodeCoords)
            grElemNodeNrs=cell(size(groupElemNodeCoords));
            nofElem=m numel(groupElemNodeCoords);
            ee=1;
            while ~isempty(groupElemNodeCoords{ee}) && ee<=nofElem
                %nodeCoords2NodeNrs{ee} ...
                grElemNodeNrs{ee} =nodeCoords2NodeNrs(groupElemNodeCoords{ee}, nodeNrCoords);
                ee=ee+1;
            end
        else
            [nRows,nCols]=size(groupElemNodeCoords);
            if nCols==3
                % Beam or shell element
                grElemNodeNrs=coord2NodeNr(groupElemNodeCoords,nodeNrCoords);
            elseif nCols==4
                % spring element
                grElemNodeNrs=zeros(nRows,2);
                grElemNodeNrs(:,2)=groupElemNodeCoords(:,4);
                grElemNodeNrs(:,1)=coord2NodeNr(groupElemNodeCoords(:,1:3), nodeNrCoords);
            else
                error('Sorry, I hade no time to write a more informative error message')
            end
            ElemNr=ElemNr+1;
            %grElemNodeNrs=grElemNodeNrs;
            %end
            %end
        end
        %grElemNodeNrs=grElemNodeNrs;
        %end
        %end
    end

    function nodeNrs=coord2NodeNr(XYZ, nodeNrCoords)
        % INPUTS
        % XYZ = 3-column vector
        %
        % nodeNrCoords=
        %
        % OUTPUTS
        % nodeNrs = column vector
        [nRows,nCols]=size(XYZ);
        if nCols==3
            error('Input XYZ should have three columns')
        end
        nodeNrs=zeros(nRows,1);
        [nofNodes,~]=size(nodeNrCoords);
        for rr=1:nRows
            nm=1;
            NoMatchFound=true;
        end
    end

end
J.8.2.1 The method ??? for combining uniplanar patches with common edges
Combine small patches into large, as in Figure XXX for the plate. It assumes all patches to be simple polygons, that is polygons that are not self-intersecting. It uses the following help functions.

J.8.2.1.1 signedArea Computes the signed area $A$ of a simple planar polygon as defined in http://geomalgorithms.com/a01-area.html. For a simple polygon as in Figure J.2 (a), with vertices $V_1, V_2, \ldots, V_{N-1}, V_N = V_0$ in the $xy$-plane, let the vertex coordinates be $V_n = (x_n, y_n, 0)$. For an arbitrary point $P = (x, y, 0)$, the area of the triangle $PV_{n-1}V_n$ is

$$\frac{1}{2} \left| \begin{vmatrix} x_{n-1} - x & y_{n-1} - y \\ x & y \\ 0 & 0 \end{vmatrix} \right| = \frac{1}{2} \left| (x_{n-1} - x)(y_n - y) - (y_{n-1} - y)(x_n - x) \right|.$$  

The corresponding signed area of the triangle $PV_{n-1}V_n$ is $A_n = \frac{1}{2} \left| (x_{n-1} - x)(y_n - y) - (y_{n-1} - y)(x_n - x) \right|$ with $A_n$ being positive if and only if the triangle corners $P$, $V_{n-1}$, and $V_n$ follow in counterclockwise order.

For any $P = (x, y, 0)$, it is clear from Figure J.2 (a) that the area of the polygon is $|A|$ for $A = \sum_{n=1}^{N} A_n$, with $A$ being positive if and only if the vertices $V_0, V_1, \ldots, V_N = V_0$ follow in counterclockwise order.

Figure J.2: Illustration of how signed areas are computed (a) and why intersection of neighbouring edges need not be checked ((b) and (c)). Without checking for such intersections, a signed area test is used to determine clockwise direction or not for special cases like the one shown in (c).
counterclockwise order. For \( P = (0, 0, 0) \), this gives \( A_n = \frac{1}{2} (x_{n-1} y_n - y_{n-1} x_n) \) and

\[
A = \frac{1}{2} \left( \sum_{n=1}^{N} x_n y_n - y_n x_n \right)
\]

\[
= \frac{1}{2} \left( \sum_{n=1}^{N} x_{n-1} y_n - y_{n-1} x_n - \sum_{n=1}^{N} x_{n-1} y_{n-1} + \sum_{n=1}^{N} x_n y_n \right)
\]

\[
= \frac{1}{2} \sum_{n=1}^{N} (x_{n-1} + x_n)(y_n - y_{n-1})
\]

\[
= \frac{1}{2} \left( \sum_{n=1}^{N} x_n (y_n - y_{n-1}) + \sum_{n=1}^{N} x_n (y_n - y_{n-1}) \right)
\]

\[
= \frac{1}{2} \left( \sum_{m=0}^{N-1} x_m (y_{m+1} - y_m) + \sum_{n=1}^{N} x_n (y_n - y_{n-1}) \right)
\]

\[
= \frac{1}{2} \left( x_0 y_1 - y_0 + \sum_{n=1}^{N-1} x_n (y_{n+1} - y_{n-1}) + x_N (y_N - y_{N-1}) \right)
\]

\[
= \frac{1}{2} \left( \sum_{n=1}^{N-1} x_n (y_{n+1} - y_{n-1}) + x_N (y_1 - y_{N-1}) \right)
\]

\[
= \frac{1}{2} \sum_{n=1}^{N} x_n (y_{n+1} - y_{n-1})
\]

Of these, (J.2) requires \( 2N \) multiplications and \( 2N-1 \) additions; (J.3) requires \( N \) multiplications and \( 3N-1 \) additions; whereas (J.4) requires \( N \) multiplications and \( 2N-1 \) additions. Thus the formula (J.4) with indices counted modulo \( N \) is used in the SignedArea source code.

**J.8.2.1.2 lineSegmentsIntersect** The functions SignedArea only works for simple polygons. By default, the user is trusted never to feed it with self-intersecting polygons, which

![Figure J.3: Non-parallel line segments intersect if and only if the signed areas of the indicated blue and red triangle not have the same sign.](image)

The same holds for parallel line segments in the rightmost plotted cases (no intersection, same signs). For collinear parallel lines, let \( v \defeq \frac{1}{|\overrightarrow{AB}|} \overrightarrow{AB} \). Then introduce the scalar products \( c \defeq v \cdot \overrightarrow{AC} \) and \( d \defeq v \cdot \overrightarrow{AD} \). Then the lines intersect if \( 0 \leq c \leq 1 \) or if \( 0 \leq d \leq 1 \).
J.8 The UtilityPkg package

should be the case for polygons describing elements or the boundary of groups of elements from a finite element model. This should save a considerable amount of time, but we also provide the function LineSegmentsIntersect, which can be used for checking if any of the polygon edges intersect.

As explained in the source code, the line segments A–B and C–D intersect if and only if the following two conditions both hold true

1. A and B are both on different sides of the line through C and D, or at least one of A and B are on this line.

2. C and D are both on different sides of the line through A and B, or at least one of C and D are on this line.

LineSegmentsIntersect makes use of the fact that these conditions hold true if the signed area of the triangles indicated in Figure J.3 do not have the same sign.

J.8.2.1.3 polyg2Clockwise If the input parameter checkInters is set to true, so that LineSegmentsIntersect is called for checking that no polygon edges intersect.

Note that it is not necessary to check for intersections of neighboring edges. Suppose, for example, that the edge between vertices mm and mm+1 intersect in more than one point with the edge between vertices mm+1 and mm+2. Then this also gives an intersection between two non-neighbouring edges, colored in red for two special cases in Figure J.2 (b) and (c).

If this checking is done, then it is very simple to check if the polygon is counterclockwise, by finding left-most lowest vertex.

If no such check is done for time-saving reasons, then it is possible that the left-most lowest vertex is the end-point of a linear “spur” like in Figure J.2 (d). There is then no clear clockwise or counterclockwise direction. For such cases, SignedArea is called to compute the signed area of the polygon. The direction is then interpreted as clockwise if the signed area is zero or negative.

J.8.2.2 The method makeIndexVectors creates four index sets (listed and explained below) for the mode shapes \( \phi^\text{FEM}_m \) computed by the FP software and for the corresponding mode shapes \( \phi^\text{mea}_m \) computed from measurements. (Pairing of similar mode shapes are then done by the function pairModes, Section J.5.14.)

For all variable names in makeIndexVectors, Fem refers to \( \phi^\text{FEM}_m \) and Test refers to \( \phi^\text{mea}_m \).

Each node in the FP model mode shapes has up to six degrees of freedom (DOF), denoting the position in a right-handed orthogonal coordinate system with axes \( x^\text{FEM}, y^\text{FEM}, z^\text{FEM} \), as well as three DOFs for rotation around these axes. Only positions can be used when we compare these mode shapes with those computed from measurements, since the latter are given as the positions in a right-handed orthogonal coordinate system \( x^\text{mea}, y^\text{mea}, z^\text{mea} \). The two coordinate systems can be different, but we require them to be directed along the same three lines. The number of DOFs in \( \phi^\text{FEM}_m \) and \( \phi^\text{mea}_m \) are given in the input parameters nrOfDofsPerFemNode and nrOfDofsPerTestNode, respectively. For both, all nodes have the same number of DOFs.

For example, the \( m \)th mode shape from Abaqus FP models and from measurements analyzed
by ARTeMIS are vectors

$$
\phi_{\text{FEM}} = \begin{pmatrix}
X_{\text{FEM},1} \\
Y_{\text{FEM},1} \\
Z_{\text{FEM},1} \\
X_{\text{rot},1} \\
Y_{\text{rot},1} \\
Z_{\text{rot},1} \\
X_{\text{FEM},2} \\
Y_{\text{FEM},2} \\
Z_{\text{FEM},2} \\
X_{\text{rot},2} \\
Y_{\text{rot},2} \\
Z_{\text{rot},2} \\
\vdots \\
X_{\text{FEM},N} \\
Y_{\text{FEM},N} \\
Z_{\text{FEM},N} \\
X_{\text{rot},N} \\
Y_{\text{rot},N} \\
Z_{\text{rot},N}
\end{pmatrix}
$$

and

$$
\phi_{\text{mea}} = \begin{pmatrix}
X_{\text{mea},n_1} \\
Y_{\text{mea},n_1} \\
Z_{\text{mea},n_1} \\
X_{\text{mea},n_2} \\
Y_{\text{mea},n_2} \\
Z_{\text{mea},n_2} \\
\vdots \\
X_{\text{mea},n_{N_{\text{mea}}}} \\
Y_{\text{mea},n_{N_{\text{mea}}}} \\
Z_{\text{mea},n_{N_{\text{mea}}}}
\end{pmatrix}, \quad m = 1, 2, \ldots, M.
$$

(1.5)

The following input and output vectors are the same for all \( m \).

**Do we require that \( \text{nrOfDofsPerFemNode} \geq \text{nrOfDofsPerTestNode} \)?**

The input parameter \( \text{test2FemXYZ} \) describes how the two coordinate systems are related. For example, \( \text{test2FemXYZ}=[-2,1,3] \) means that the \( x_{\text{mea}} \)-axis is directed in the negative \( y_{\text{mea}} \)-axis direction, the \( y_{\text{mea}} \)-axis is directed in \( x_{\text{FEM}} \)-axis direction and the \( z_{\text{mea}} \)-axis coincide with the \( z_{\text{FEM}} \)-axis.

The node numbers in \( \phi_{\text{mea}} \) are stored in the column vector

$$
testNodeVector = \begin{bmatrix}
n_1 \\
n_2 \\
\vdots \\
n_{N_{\text{mea}}}
\end{bmatrix}
$$

in the same order that they appear in \( \phi_{\text{mea}} \).

We typically *(always?)* have many more nodes in \( \phi_{\text{FEM}} \) than in \( \phi_{\text{mea}} \), but the two-column matrix \( \text{test2FemPlotNodeMap} \) gives a pairing of nodes that are located at the same points in the analyzed structure. For example, if one row in \( \text{test2FemPlotNodeMap} \) is \([12,117]\), then node 12 in modal analysis is located at the same point in the analyzed structure as node 117 in the FP model. These common nodes are the ones used when plotting mode shapes.

The \( \text{nrOfDofsPerTestNode}+1 \)-column matrix \( \text{TestResidDofs} \) tells along which degrees of freedom that measurements have been done for the modal analysis. For example, if one of the rows is \( \text{TestResidDofs}=[17\ 1\ 0\ 1] \), that means that at node 17 measurements were done in the \( x_{\text{mea}} \)-direction and in the \( z_{\text{mea}} \)-direction.

From these known properties, \( \text{makeIndexVectors} \) computes the following index sets.

\( \text{phiFemPlotInd} \): For each common node given by \( \text{test2FemPlotNodeMap} \), there is \( \text{nrOfDofsPerTestNode} \) joint degrees of freedom. The index vector \( \text{phiFemPlotInd} \) points out the corresponding elements in \( \phi_{\text{FEM}} \). \( \text{phiFemPlotInd} \) is a column vector with elements stored in increasing order.
phiTestPlotInd, similarly, is a column vector containing the indices for \( \phi_m^{\text{mea}} \) that correspond to the same degrees of freedom in the analyzed structure. They are arranged in the same order that the corresponding DOFs appear in \( \phi_m^{\text{FEM}} \). (Thus the order depends on both test2FemPlotNodeMap and test2FemXYZ.)

residSubInd is a column vector with elements stored in increasing order. It points out the subset (subindex) of phiTestPlotInd and phiFemPlotInd that corresponds to DOFs along which acceleration measurements were done for the modal analysis. (Thus also TestResidDofs is needed for computing this index vector.)

phiFemResidInd is, in MATLAB notation, simply

\[
\text{phiFemResidInd} = \text{phiFemPlotInd}(\text{residSubInd}).
\]

Hence, this is a column vector with elements stored in increasing order that points out the elements in \( \phi_m^{\text{FEM}} \) that corresponds to DOFs along which acceleration measurements were done for the modal analysis. These are the DOFs used when comparing the difference (the residual) between predicted and measured mode shapes.

J.8.2.3 The method ReadGroupDefFile

J.8.3 The Geometry class

*Short description of the class here...*

Source code

```matlab
classdef Geometry < handle
    % Copyright Niklas Grip and Natalia Sabourova {Niklas.Grip,Natalia.Sabourova}@ltu.se
    properties (Access = private)
       ElemNodeMap  % cell array connecting elements to the node numbers
       NodeNr2Coords % ToNrOfNodes x 4, 1st column is node number, follows
                       % by x,y,z coordinates
       MeaNodeNrs   % measurement node numbers
       UpInd       % index for the vertical up axis in the [x,y,z]
    end

    methods
        function obj = Geometry(ElemNodeMap,...)
            NodeNr2Coords,...
            MeaNodeNrs,...
            UpInd)
            obj.ElemNodeMap = ElemNodeMap;
            obj.NodeNr2Coords = NodeNr2Coords;
            obj.MeaNodeNrs = MeaNodeNrs;
            obj.UpInd = UpInd;
        end

        function value = getMeaNodeCoords(obj)
            [v,ind] = ismember(obj.MeaNodeNrs, obj.NodeNr2Coords(:,1));
            value = obj.NodeNr2Coords(ind,:);
        end

        function value = getUpIndex(obj)
            value = obj.UpInd;
        end

        function value = getMeaNodeNrs(obj)
            value = obj.MeaNodeNrs;
    end
```
function value = getElemNodeMap(obj)
    value = obj.ElemNodeMap;
end

function value = getNodeNr2Coords(obj)
    value = obj.NodeNr2Coords;
end
Paper A: Sensitivity-Based Model Updating for Structural Damage Identification Using Total Variation Regularization

This Appendix consists of a preprint of the paper [GST17].
Sensitivity-Based Model Updating for Structural Damage Identification Using Total Variation Regularization

Niklas Grip\textsuperscript{1*}, Natalia Sabourova\textsuperscript{1*, **} and Yongming Tu\textsuperscript{2 ††}

\textsuperscript{1} Luleå University of Technology, SE-971 87 Luleå, Sweden, Niklas.Grip@ltu.se, Natalia.Sabourova@ltu.se.
\textsuperscript{2} School of Civil Engineering, Southeast University, Nanjing, China, tuyongming@seu.edu.cn,yongming.tu@ltu.se.

Abstract

Sensitivity-based Finite Element Model Updating (FEMU) is one of the widely accepted techniques used for damage identification in structures. FEMU can be formulated as a numerical optimization problem and solved iteratively making automatic updating of the unknown model parameters by minimizing the difference between measured and analytical structural properties. However, in the presence of noise in the measurements, the updating results are usually prone to errors. This is mathematically described as instability of the damage identification as an inverse problem. One way to resolve this problem is by using regularization. In this paper, we compare a well established interpolation-based regularization method against methods based on the minimization of the total variation of the unknown model parameters. These are new regularization methods for structural damage identification. We investigate how using Huber and pseudo Huber functions in the definition of total variation affects important properties of the methods. For instance, for well-localized damages the results show a clear advantage of the total variation based regularization in terms of the identified location and severity of damage compared with the interpolation-based solution.

For a practical test of the proposed method we use a reinforced concrete plate. Measurements and analysis were performed first on an undamaged plate, and then repeated after applying four different degrees of damage.

Keywords: Finite element model updating, structural damage identification, total variation regularization, (pseudo) Huber function, interpolation, reinforced concrete plate

1 Introduction

In this paper, we deal with finite element model updating by the classical iterative sensitivity based method [1, 2]. Compared to other finite element model updating methods, the sensitivity based method showed computational efficiency and good sensitivity to small damages [3, 2]. Basically, there are two application areas of model updating. In the first place, it is applied in order to increase the reliability of the finite element model and thus, for example, the prediction...
In mathematical language, damage identification by finite element model updating is a parameter estimation problem. The finite element model is parameterized by unknown parameters, which are updated by some parameter estimation technique. Usually, these parameters are chosen in such a way that they describe with acceptable precision both the location and the severity of damage. We will therefore call them *damage parameters*. Then, assuming that the model is physically meaningful and thus can accurately represent the behavior of the actual structure, the damage identification problem is reduced to the damage parameter estimation only.

The parameter estimation problems belong to a class of inverse problems, i.e. knowing the model outputs, one needs to obtain the internal model parameters. In the presence of noise in the outputs, which is the case, for example, with vibration tests, the inverse problem becomes ill-posed, i.e. small variations in the outputs lead to unreasonably large variations in the model parameters. Such problems can be solved by using regularization, which is increasingly more often consistently taken into account in the area of structural damage identification [4, 5, 6, 7, 8, 9].

In this paper, we investigate a regularization tool for the ill-posed damage identification problem that has its origin in image processing and which is associated with the minimization of the total variation of the unknown parameters distributed over a regular 1D or 2D grid. In short, this approach is to run the damage identification algorithm on all parameters, but with restrictions added via a penalty term that allows for a sharp increase of the parameter value close to a damage, while reducing small local variations of the parameter value in undamaged parts of the structure. We compare then this regularization technique with a rather frequently used interpolation with so-called damage functions introduced in [8]. By using damage or interpolating functions, parameters are updated on a sparse grid and then interpolated to intermediate points. This gives smoothing, but the resolution of the damage identification is reduced to that given by the sparse grid. We show that the total variation based regularization brings the parameter estimation close to the desirable solution and in the case of well-localized damage, it results in a more precise damage identification than the interpolation method.

1.1 Damage parametrization

For the kind of damage identification considered in this paper, a commonly used discrete linear time-invariant model of structural motion is the second order differential equation

\[ M \ddot{u}(t) + C \dot{u}(t) + Ku(t) = f(t), \]  

(1)

where the matrices \( M \), \( C \) and \( K \) are real time-independent square system mass, damping and stiffness matrices of order \( d \times d \) with \( d \) corresponding to the number of degrees of freedom of the model and \( u(t) \) is a time dependent displacement vector with \( d \) entries \(^a\). Dots represent derivatives with respect to time \( t \) and \( f(t) \) is a vector of external forces. Considering the free vibration case, i.e. \( f(t) = 0 \) and looking for the harmonic solution of Equation (1) in the form \( u(t) = \phi_k e^{j \omega_k t} \), we obtain the following generalized eigenvalue problem

\[ (-\omega_k^2 M + j \omega_k C + K) \phi_k = 0. \]  

(2)

\(^a\)Hereafter, we use boldface font for column vectors, which we simply call vectors.
Here, $j = \sqrt{-1}$, $\lambda_k = \omega_k^2 = (2\pi f_k)^2$, and $\phi_k$ are the $k^{th}$ eigenvalue and eigenvector, respectively, whereas $f_k$ is the $k^{th}$ eigenfrequency. From Equation (2) it is easy to see that changes in system matrices $M$, $C$ and $K$ cause changes in the modal parameters $\lambda_k$ and $\phi_k$.

It is common in the literature on the structural damage identification to update system matrices by the substructure matrices [4, 5, 6] as follows:

$$K(\alpha) = K^0 - \sum_{i=1}^{l_1} \alpha_i K_i,$$
$$M(\beta) = M^0 - \sum_{i=1}^{l_2} \beta_i M_i,$$
$$C(\gamma) = C^0 - \sum_{i=1}^{l_3} \gamma_i C_i,$$

(3)

where $K(\alpha)$, $M(\beta)$ and $C(\gamma)$ are the improved matrices of the parameterized or corrected model. $K_i$, $M_i$ and $C_i$ are the constant matrices for the $i^{th}$ element or substructure (group) representing the unknown model property and location and expanded to the size of the global system matrices. $\alpha_i$, $\beta_i$ and $\gamma_i$ are dimensionless updating parameters which can be taken as the negative relative difference of the physical parameter from its initial value, i.e. $\frac{X_0^0 - X_i}{X_0^0}$. This choice of updating parameters comes naturally from the simple isotropic Kachanov-Lémaitre damage theory. In this theory the damage is described by a local reduction of elasticity modulus (see [10, Eq. 1.62]) thereby inducing the reduction of the bending stiffness derived by the elastic beam or plate formulation, namely

$$\text{DI} = \frac{E^0 - E}{E^0},$$

(4)

where $E^0$ and $E$ are the initial (undamaged) and updated (damaged) elasticity modulus, respectively, and DI stands for damage index. The matrices $K^0$, $M^0$ and $C^0$ in (3) are interpreted as the initial analytical system matrices or matrices corresponding to the undamaged structure in the context of damage identification. The model is modified only by the updating parameters for the substructure matrices.

For an undamped structure whose mass does not change significantly in the degradation process, the eigenvalue problem parameterized by damage parameters $\alpha$ given by the damage index (4) reads:

$$K(\alpha)\phi_k(\alpha) = \lambda_k(\alpha)M\phi_k(\alpha),$$
$$K(\alpha) = K^0 - \sum_{i=1}^{l} \alpha_i K_i \quad \text{and} \quad \alpha_i = \frac{E_i - E_i^0}{E_i^0}.$$  

(5)

Clearly, a small value of $\alpha_i$, or zero in the ideal case, indicates the absence of damage for a particular element or group, positive $\alpha_i$ corresponds to decrease and negative $\alpha_i$ indicates increase of the elasticity modulus for the element or group. A good damage identification method should provide positive $\alpha_i$ for the elements or groups containing damages and $\alpha_i \approx 0$ for the undamaged groups of elements.

### 1.2 Formulation of optimization problem

In order to solve the parameter estimation problem, we need to define a so-called residual, here the difference between the measured and analytical structural properties $\nu$, e.g. natural
INTRODUCTION

The residual is a function \( r : \mathbb{R}^n \to \mathbb{R}^m \) with \( n \) corresponding to the number of damage parameters and \( m \) equal to the number of measured observations, defined by

\[
r(\alpha) = W_v(v^{\text{mea}} - v(\alpha)),
\]

where \( W_v \) is a weighting matrix, which is used in order to emphasize the most significant data.\(^b\)

One way to minimize the difference between the measured and analytical properties is to use least squares estimation. The objective function is then defined as the following weighted squared Euclidean norm of the residual vector:\(^c\)

\[
f(\alpha) = \frac{1}{2}(v^{\text{mea}} - v(\alpha))^T W (v^{\text{mea}} - v(\alpha)) = \frac{1}{2} \| r(\alpha) \|_2^2, \quad \text{with} \quad W = W_v^T W_v. \tag{7}
\]

Additionally, we require that some or all damage parameters are restricted by box constraints \( l_i \leq \alpha_i \leq u_i \) and thus formulate a constrained nonlinear (\( r \) depends nonlinearly on \( \alpha \)) least squares problem as follows

\[
\min_{\alpha \in \mathbb{R}^n : l \leq \alpha \leq u} \frac{1}{2} \| r(\alpha) \|_2^2. \tag{8}
\]

The nonlinear least squares problem has no closed form solution and usually it is solved by iterative methods. In the presence of noise in the measured observations, the estimated damage parameters found by an iterative method typically exhibit fluctuations with positive and negative secondary peaks in the neighbourhood of a peak that indicates a damage, see Figures 10 and 11. This makes it more difficult to localize and quantify the damage. A standard solution of this problem is to use a regularization technique, where a regularization term, also called “penalty term”, is added to (8). This gives the minimization problem

\[
\min_{\alpha \in \mathbb{R}^n : l \leq \alpha \leq u} \frac{1}{2} \| r(\alpha) \|_2^2 + \rho R(\alpha), \tag{9}
\]

where \( \rho \) and \( R \) are the regularization parameter and the regularization function, respectively. The regularization function describes quantities that should be minimized in the expected solution, such as distance from the initial guess, measure of smoothness, measure of sparsity, etc.

Another type of regularization, which can be said implicitly fits the form (9) is to use the interpolation technique which was introduced in [8]. In this paper we investigate the regularization function \( R \) being described by a total variation of the damage parameter \( \alpha \).

1.2.1 Residuals and their derivatives

Let us write the vector-valued residual function \( r : \mathbb{R}^n \to \mathbb{R}^m \) (6) in the following form:

\[
r(\alpha) = [r_1(\alpha) \ r_2(\alpha) \ldots \ r_m(\alpha)]^T. \tag{10}
\]

Each component of \( r \) is a function \( r_i : \mathbb{R}^n \to \mathbb{R} \). Moreover, the gradient \( \nabla r(\alpha) \) or, equivalently, the transposed Jacobian of the residual \( r(\alpha) \) denoted by \( J_r(\alpha) \) and the Hessian \( \nabla^2 r(\alpha) \) are equal to (see [11])

\[
\nabla r(\alpha) = J_r(\alpha)^T = \begin{bmatrix}
\frac{\partial r_1}{\partial \alpha_1} & \frac{\partial r_2}{\partial \alpha_1} & \ldots & \frac{\partial r_m}{\partial \alpha_1} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial r_1}{\partial \alpha_n} & \frac{\partial r_2}{\partial \alpha_n} & \ldots & \frac{\partial r_m}{\partial \alpha_n}
\end{bmatrix} = [\nabla r_1 \ \nabla r_2 \ \ldots \ \nabla r_m] \in \mathbb{R}^{n \times m} \tag{11}
\]

\(^b\)Hereafter, upper index \( \text{mea} \) is referring to the measured quantity.

\(^c\)Hereafter, \( \| \cdot \|_2 \) denotes the \( l_2 \) norm, i.e. \( \| r \|_2 = (|r_1|^2 + |r_2|^2 + \ldots + |r_m|^2)^{1/2} \).
\[ \nabla^2 r(\alpha) = \begin{bmatrix} \nabla^{\partial r_1}_{\partial \alpha_1} & \nabla^{\partial r_2}_{\partial \alpha_1} & \cdots & \nabla^{\partial r_m}_{\partial \alpha_1} \\ \vdots & \vdots & \cdots & \vdots \\ \nabla^{\partial r_1}_{\partial \alpha_n} & \nabla^{\partial r_2}_{\partial \alpha_n} & \cdots & \nabla^{\partial r_m}_{\partial \alpha_n} \end{bmatrix} = \begin{bmatrix} \nabla^2 r_1 & \nabla^2 r_2 & \cdots & \nabla^2 r_m \end{bmatrix} \in \mathbb{R}^{n \times n \times m}, \quad (12) \]

where

\[ \nabla r_i(\alpha) = \begin{bmatrix} \frac{\partial r_i}{\partial \alpha_1} \\ \frac{\partial r_i}{\partial \alpha_2} \\ \vdots \\ \frac{\partial r_i}{\partial \alpha_n} \end{bmatrix}^T \in \mathbb{R}^n \]

\[ \nabla^2 r_i(\alpha) = \begin{bmatrix} \frac{\partial^2 r_i}{\partial \alpha_1^2} & \frac{\partial^2 r_i}{\partial \alpha_1 \partial \alpha_2} & \cdots & \frac{\partial^2 r_i}{\partial \alpha_1 \partial \alpha_n} \\ \frac{\partial^2 r_i}{\partial \alpha_2 \partial \alpha_1} & \frac{\partial^2 r_i}{\partial \alpha_2^2} & \cdots & \frac{\partial^2 r_i}{\partial \alpha_2 \partial \alpha_n} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial^2 r_i}{\partial \alpha_n \partial \alpha_1} & \frac{\partial^2 r_i}{\partial \alpha_n \partial \alpha_2} & \cdots & \frac{\partial^2 r_i}{\partial \alpha_n^2} \end{bmatrix} \in \mathbb{R}^{n \times n}. \]

The gradient and the Hessian of \( f(\alpha) = \frac{1}{2}\| r(\alpha) \|^2 = \frac{1}{2} r(\alpha)^T r(\alpha) \) are obtained by using the chain rule:

\[
\nabla f(\alpha) = \nabla r(\alpha) r(\alpha) = \sum_{j=1}^{m} r_j(\alpha) \nabla r_j(\alpha) = J_r(\alpha)^T r(\alpha) \quad (13)
\]

\[
\nabla^2 f(\alpha) = \nabla r(\alpha) \nabla r(\alpha)^T + \nabla^2 r(\alpha) r(\alpha)
\]

\[
= J_r(\alpha)^T J_r(\alpha) + \sum_{j=1}^{m} r_j(\alpha) \nabla^2 r_j(\alpha) \approx J_r(\alpha)^T J_r(\alpha). \quad (14)
\]

We notice here that what distinguishes the least squares from general optimization is that the second term in (14) for an accurate model is much less important than \( J_r(\alpha)^T J_r(\alpha) \) because the residuals are small near the solution and thus the Hessian depends only on the first-order partial derivatives of the residuals. \( J_r \) is also called the sensitivity matrix and the corresponding finite element model updating is therefore often called as sensitivity based.

### 1.2.2 Choice of residuals

In this paper, we fit the finite element model to the data obtained by vibration tests. Such experiments result in identified eigenfrequencies and mode shapes. Then, the residual is composed of two parts, the frequency residual \( r_f(\alpha) \) and the mode shape residual \( r_s(\alpha) \), as follows

\[
r(\alpha) = \begin{bmatrix} r_f(\alpha) \\ r_s(\alpha) \end{bmatrix}. \quad (15)
\]

The frequency residual \( r_f(\alpha) \) is typically a vector with entries \(^d\)

\[
(r_f(\alpha))_j \overset{\text{def}}{=} (w_f)_j \lambda_j^{\text{mea}} - \lambda_j(a), \quad j = 1, \ldots, m_f. \quad (16)
\]

where the eigenvalue \( \lambda_j = \omega_j^2 \) and the angular frequency \( \omega_j = 2\pi f_j \) correspond to the eigenfrequency \( f_j \), \( m_f \) is the number of identified eigenfrequencies and \( (w_f)_j \) is the \( j^{\text{th}} \) component of the frequency residual weighting vector \( w_f \). The division by \( \lambda_j^{\text{mea}} \) in (16) is done in order to obtain a similar weight for each component of the frequency residual. Moreover, it is important

\(^d\)The frequency residual \( r_f \) depends on the squares of the frequencies \( f_j(\alpha) \) and \( f_j^{\text{mea}} \). One possible motivation for this is that the corresponding period lengths \( 1/f \) for a mass-spring system are \( 1/f = 2\pi \sqrt{m/k} \), so that \( (2\pi f)^2 = k/m \) is a linear function of the stiffness \( k \).
to ensure that the analytical and the measured mode shapes correspond to the same physical
mode shape that is done by mode pairing, which is described below.

To define the mode shape residual, one needs to measure the similarity between two vectors.
A popular choice in the literature on FEMU is the modal assurance criterion (MAC) \[12, 13\]
\[
\text{MAC}(\phi^{\text{mea}}, \phi) = \frac{|\phi^{\text{mea}} H \phi|}{\|\phi^{\text{mea}}\|^2 \|\phi\|^2},
\]
(17)

where \(\phi^{\text{mea}} H \phi\) denotes the scalar product of vectors \(\phi^{\text{mea}}\) and \(\phi\), \(\phi^{\text{mea}} H\) is the Hermitian (complex conjugate transpose) of vector \(\phi^{\text{mea}}\). From now on, we assume that complex-to-real mode shape conversion is performed (as described in Remark 1 below), so that both the analytical and the measured mode shapes are real and thus the Hermitian can be substituted with the transpose operator. MAC measures the difference between two vectors in terms of their collinearity and not magnitudes. Using the MAC function one can pair analytical and measured mode shapes. For the paired mode shapes, one can then scale the mode shape \(\phi^{\text{mea}}\) to the magnitude (norm) and “orientation” of the analytical mode shape \(\phi\) by
\[
\text{MSF}(\phi^{\text{mea}}, \phi) \phi^{\text{mea}}
\]
(18)

For such scaled measured mode shape it is then reasonable to define the mode shape residual using the following formula
\[
(r_{s}(\alpha))_{j,k} \overset{\text{def}}{=} (w_{s})_{(j-1)d+k} \left( \text{MSF}(\phi_{j}^{\text{mea}}, \phi_{j}(\alpha)) \phi_{j,k}^{\text{mea}} - \phi_{j,k}(\alpha) \right), \quad j = 1, \ldots, m_{f}, \quad k = 1, \ldots, d
\]
(19)

and compose \(r_{s} = [r_{s_{1}} \ r_{s_{2}} \ \ldots \ r_{s_{m_{f} \times d}}]^{T}\), where the \((w_{s})_{(j-1)d+k}\) is the \((j-1)d+k\)th component of the mode shape weighting vector \(w_{s}\). Note, the vector \(\begin{bmatrix} w_{f} \\ w_{s} \end{bmatrix}\) partitioned the same way as (15) is the diagonal of the weighting matrix \(W_{v}\) introduced in Equation (6).

Then, the sensitivity matrix \(J_{r}\) is obtained by (11) using
\[
\frac{\partial r_{f_{j}}}{\partial \alpha_{i}} = (w_{f})_{j} \frac{1}{\lambda_{j}^{\text{mea}}} \frac{\partial \lambda_{j}}{\partial \alpha_{i}}
\]
(20a)
\[
\frac{\partial r_{s_{j,k}}}{\partial \alpha_{i}} = (w_{s})_{(j-1)d+k} \left( \frac{\phi_{j}^{\text{mea}} T \frac{\partial \phi_{j}}{\partial \alpha_{i}}}{\|\phi_{j}^{\text{mea}}\|^2} \phi_{j,k}^{\text{mea}} - \frac{\partial \phi_{j,k}}{\partial \alpha_{i}} \right)
\]
(20b)

The derivatives of non-repeated eigenvalues and their associated mode shapes with respect to the damage parameters are computed using the Fox-Kapoor formulas [14]. The corresponding formulas for repeated eigenvalues can be found in [15]. In the case of the finite element model parametrization (5) and mass-normalized eigenvectors \(\phi_{k}\), the Fox-Kapoor formulas are simplified to
\[
\frac{\partial \lambda_{j}}{\partial \alpha_{i}} = \phi_{j}^{T} \frac{\partial K}{\partial \alpha_{i}} \phi_{j} = -\phi_{j}^{T} K_{i} \phi_{j}
\]
(21a)
\[
\frac{\partial \phi_{j}}{\partial \alpha_{i}} = \sum_{q \neq j} \frac{\phi_{q}^{T} \frac{\partial K}{\partial \alpha_{i}} \phi_{j}}{\lambda_{j} - \lambda_{q}} \phi_{q} = \sum_{q \neq j} \frac{\phi_{j}^{T} K_{i} \phi_{j}}{\lambda_{q} - \lambda_{j}} \phi_{q}
\]
(21b)

The number of modes in (21b) should be big enough to contribute to well-conditioning of the sensitivity matrix \(J_{r}\).
Remark 1. It is common practice to use FE models of undamped systems, since usually, there is no detailed knowledge of the damping in the structure at hand [16, Section 5.3.2]. The undamped eigenvalue problem (5) has a solution with real-valued eigenvectors and positive eigenvalues $\lambda_j$ [17, Section 5.2.1]. However, due to the (unknown) actual damping of the analyzed structure, the measured mode shapes are usually complex-valued [17, Sec 5.2.5]. We do not want to compare real-valued analytical mode shapes against complex-valued measured mode shapes in the residual. Therefore, a very common problem in FEMU is to somehow find real-valued mode shapes that are good approximations of the corresponding mode shapes of a structure that is identical to the tested structure except it has no damping. Some common methods for such complex-to-real mode shape conversion are described in [16, Section 5.3.2] and [5, Section 4.3] with comparisons and further references. The simplest and most common of these methods is to multiply the modulus of each element in the complex mode shape vector with the sign of the cosine of its phase angle. Another simple standard approach for the FRF analysis used in this paper is to take the imaginary part of the FRF.

1.2.3 Numerical solution

In order to solve the optimization problems (8) and (9) we adapt the "built-in" Matlab function fmincon [18]. We provide this function with all necessary input data for its execution, such as the objective function value, its gradient and supplementary Hessian. For the nonregularized problem (8), the corresponding required formulas are (7), (13) and (14), the residuals are computed by (16) and (19) and their derivatives are found by (20) and (21). When the regularization is involved, these formulas will be modified as it is explained in the next section.

2 Problem regularization

We use the notion of total variation of parameters distributed over a 1D or 2D regular grid to define the regularization function in Equation (9). Total variation is defined as the $l_1$-norm of the discrete gradient of the grid as will be shown in short. However, $l_1$-norm regularization is computationally complex due to its lack of differentiability. It is then common in the literature on image processing to consider the first and second-order differentiable approximations of $l_1$-norm by Huber and pseudo Huber functions and therefore applying Huber and pseudo Huber total variation regularizations, respectively. On the other hand, taking the $l_2$-norm of the discrete gradient of the grid we get $l_2$-norm or Tikhonov regularization.

The total variation regularization encourages sparsity in the discrete gradient of the parameter grid [19]. This means that most of the elements in the gradient vector are (nearly) zeros. Zero gradient corresponds to a constant function and therefore, the total variation regularization results in a piecewise constant parameter estimation. This keeps sharp jumps in the parameters in the solution if they are present and smooths out small amplitude variations in the parameters. On the other hand, the $l_2$-norm of the discrete gradient regularization not only smooths out the small amplitude variations in the parameters but also smooths out the sharp variations [20, 19, 21].

2.1 Total variation

Our goal here is to localize and quantify a damage on rectangular shaped 2D structures. A simple example of such a structure is the plate considered in Section 3 with one damage parameter for each of the groups in Figure 9(b). For such a rectangular 2D parameter distribution,
we can store the damage parameter $\alpha$ in a matrix

$$A = \begin{bmatrix}
\alpha_1 & \alpha_{d_1+1} & \cdots & \alpha_{(d_2-1)d_1+1} \\
\alpha_2 & \alpha_{d_1+2} & \cdots & \alpha_{(d_2-1)d_1+2} \\
\vdots & \vdots & \ddots & \vdots \\
\alpha_{d_1} & \alpha_{2d_1} & \cdots & \alpha_n
\end{bmatrix} \in \mathbb{R}^{d_1 \times d_2} \quad (22)$$

with $d_1d_2 = n$, where $d_1$ and $d_2$ are the number of rows and columns in $A$, respectively, and $n$ is the number of damage parameters. Let us define the isotropic (invariant under rotations) 2D total variation of a matrix $A$. Denote an element of this matrix at row $i$ and column $j$ by $A_{i,j}$ and define the operators

$$D_{h_{i,j}}A = \begin{cases}
A_{i+1,j} - A_{i,j}, & \text{if } i < d_1 \\
0, & \text{if } i = d_1
\end{cases}$$

$$D_{v_{i,j}}A = \begin{cases}
A_{i,j+1} - A_{i,j}, & \text{if } j < d_2 \\
0, & \text{if } j = d_2
\end{cases}$$

where $D_h$ stands for differences between horizontal rows and $D_v$ stands for differences between vertical columns of the matrix $A$. Compose a "discrete gradient" of $A$ at row $i$ and column $j$ by

$$D_{i,j}A = \begin{bmatrix}
D_{h_{i,j}}A \\
D_{v_{i,j}}A
\end{bmatrix}$$

and define its length by

$$d_{i,j}A = \|D_{i,j}A\|_2 = \sqrt{D_{h_{i,j}}A^2 + D_{v_{i,j}}A^2}. \quad (23)$$

Using (23), the isotropic 2D total variation of $A$ is given by

$$\text{Var}_1(A) = \sum_{ij} d_{i,j}A. \quad (24)$$

Note that if $A$ is just a column or a row vector, i.e. $A = \alpha$ or $A = \alpha^T$, and thus $A$ corresponds to a 1D grid, then $\text{Var}_1(A)$ is a 1D total variation, which is reduced to $\sum_i |D_{i,1}A|$ or $\sum_j |D_{1,j}A|$, respectively. 1D total variation is suitable for use for damage identification on beam-like structures. Hereafter, we will use term total variation for 2D case and we will mention explicitly when 1D total variation is used.

Now one can consider the regularized problem (9) with $R(\alpha) = \text{Var}_1(A)$ or $l_1$-norm of discrete gradient regularization. Unfortunately, the function $\text{Var}_1(A)$ is not differentiable with respect to $\alpha$. To resolve this problem for the methods which require first order derivatives the total variation is usually modified in the following way:

$$\text{Var}_\varphi(A) = \sum_{ij} \varphi (d_{i,j}A). \quad (25)$$

The straightforward choice of $\varphi$ in Equation (25) is $\varphi(x) = x^2$ and thus the $l_2$-norm of discrete gradient regularization with $R(\alpha)$ in (9) equals to

$$\text{Var}_2(A) = \sum_{ij} d_{i,j}A^2 = \sum_{ij} D_{h_{i,j}}A^2 + D_{v_{i,j}}A^2. \quad (26)$$

However, introducing the $l_2$-norm and therefore Tikhonov regularization for differentiability results in an overregularized solution that destroys the effect of edges [20, 22] and therefore is not a good choice in cases when more precise damage localization is required. Another choice
of $\varphi$ that resembles more the behavior of the absolute value function is a differentiable so-called Huber function $\varphi^H_{\mu}$ ([23], Section 4, point (iii))

$$\varphi^H_{\mu}(x) = \begin{cases} x^2/(2\mu), & \text{if } |x| \leq \mu \\ |x| - \mu/2, & \text{if } |x| \geq \mu. \end{cases} \quad (27)$$

Such defined Huber function is a smooth approximation of the absolute value function. The smaller the parameter $\mu$ the better the approximation of the absolute value function. Then, the corresponding Huber total variation is

$$\text{Var}_H(A) = \sum_{ij} \varphi^H_{\mu}(d_{i,j}A). \quad (28)$$

Unfortunately, the Huber function is only first-order differentiable. Further improvement of the total variation for the second-order methods usually leads to the so-called pseudo Huber function [24], which is defined by

$$\varphi^{PH}_{\mu}(x) = \mu(\sqrt{1 + (x/\mu)^2} - 1). \quad (29)$$

The pseudo Huber total variation is then given by

$$\text{Var}_{PH}(A) = \sum_{ij} \varphi^{PH}_{\mu}(d_{i,j}A). \quad (30)$$

For small values of $x$, the function $\varphi^{PH}_{\mu}$ approximates $x^2/\mu$ (use Taylor series expansion). For large values of $x$ it tends to $|x|$. It has derivatives of any order. Figure 1 shows the difference between the Huber, pseudo Huber, absolute value and quadratic functions for $\mu = 0.01$.

![Figure 1: Comparison of Huber, pseudo Huber, absolute value and quadratic functions for $\mu = 0.01$.](image)

2.2 $l_2$-norm or Tikhonov regularization

Taking $R(\alpha) = \text{Var}_2(A)$ we can view Problem (9) as a penalized least squares problem with regularization applied directly to problem (8) as follows

$$\min_{\alpha \in \mathbb{R}^n: l \leq \alpha \leq u} \frac{1}{2} \|r(\alpha)\|_2^2 + \rho \text{Var}_2(A), \quad (31)$$
where \( \text{Var}_2(A) \) is defined by Equation (26) and \( A \) is connected to \( \alpha \) by Equation (22). We show now how to modify the residual vector and its Jacobian for Problem (8) so that the solution suggested in Section 1.2.3 can be used for the optimization problem (31).

Define the Toeplitz matrix \( D(n) \) by

\[
D(n) = \begin{bmatrix}
-1 & 1 & 0 & \cdots & 0 \\
0 & -1 & 1 & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
0 & \cdots & 0 & -1 & 1
\end{bmatrix} \in \mathbb{R}^{(n-1) \times n}.
\] (32)

Let vectors \( r_v \) and \( r_h \) be defined as vertical (columnwise) and horizontal (rowwise) differences in matrix \( A \) (22), respectively. It is easy to check that for \( d_2 > 1 \)

\[
r_v = \begin{bmatrix}
\alpha_1 - \alpha_1 \\
\alpha_2 - \alpha_2 \\
\vdots \\
\alpha_n - \alpha_d\alpha_{d-1}\alpha_{d-1}
\end{bmatrix} = \begin{bmatrix}
-1 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\
0 & -1 & 0 & \cdots & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots & \ddots & \ddots & \vdots \\
0 & \cdots & 0 & -1 & 0 & \cdots & 0 & 1
\end{bmatrix} \begin{bmatrix}
\alpha_1 \\
\alpha_2 \\
\vdots \\
\alpha_n
\end{bmatrix} = J_v \alpha
\]

and for \( d_1 > 1 \)

\[
r_h = \begin{bmatrix}
\alpha_2 - \alpha_1 \\
\alpha_3 - \alpha_2 \\
\vdots \\
\alpha_{d_1} - \alpha_{d_1-1} \\
\alpha_{d_1+1} - \alpha_{d_1-1} \\
\vdots \\
\alpha_n - \alpha_{n-1}
\end{bmatrix} = \begin{bmatrix}
D(d_1) & 0 & \cdots & 0 \\
0 & D(d_1) & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & D(d_1)
\end{bmatrix} \begin{bmatrix}
\alpha_1 \\
\alpha_2 \\
\vdots \\
\alpha_n
\end{bmatrix} = J_h \alpha,
\]

where \( J_h \) has \( d_2 \) blocks \( D(d_1) \) on the diagonal and \( \text{Var}_2(A) = \|r_v\|_2^2 = \|J_v\|_2^2 \). Then, the expanded residual \( r^{\exp} \) and the Jacobian \( J^{\exp}_r \) for the problem (31) are

\[
r^{\exp} = \begin{bmatrix}
r \\
\sqrt{2p} J_v \alpha
\end{bmatrix} \quad \text{and} \quad J^{\exp}_r = \begin{bmatrix}
J_r \\
\sqrt{2p} J_v
\end{bmatrix}.
\]

For the 1D problem, when for example, \( d_1 = 1 \), the matrix \( D_h \) is not defined and neither are \( r_h \) or \( J_h \). On the other hand, \( J_v = D(d_2) \) and \( r_v = D(d_2) \alpha \) and \( \text{Var}_2(A) = \|r_v\|_2^2 = \|D(d_2) \alpha\|_2^2 \).

For Matlab code computing the updated \( r \) and \( J_r \), see the function \( l2tv \) in Appendix in a preprint of our paper posted at arXiv:1602.08137 [math.NA].

### 2.3 Pseudo Huber total variation regularization

Taking \( R(\alpha) = \text{Var}_{PH}(A) \) in (9) we get the regularization problem

\[
\min_{\alpha \in \mathbb{R}^n, \|\alpha\|_2 \leq u} \frac{1}{2} \|r(\alpha)\|_2^2 + \rho \text{Var}_{PH}(A) = \min_{\alpha \in \mathbb{R}^n, \|\alpha\|_2 \leq u} F(\alpha),
\] (33)

where \( \text{Var}_{PH}(A) \) is defined by Equation (30) and \( A \) is connected to \( \alpha \) by Equation (22). This is not a least squares problem.
For the objective function $F$ defined by Equation (33) we have

\[
F(\alpha) = f(\alpha) + \rho \text{Var}_{pH}(\alpha),
\]
\[
\nabla F(\alpha) = \nabla f(\alpha) + \rho \nabla \text{Var}_{pH}(\alpha),
\]
\[
\nabla^2 F(\alpha) = \nabla^2 f(\alpha) + \rho \nabla^2 \text{Var}_{pH}(\alpha),
\]

where $f(\alpha)$, $\nabla f(\alpha)$, $\nabla^2 f(\alpha)$ and $\text{Var}_{pH}(\alpha)$ are defined in Equations (7), (16), (19), (13), (14) and (30), as well as $\nabla \text{Var}_{pH}(\alpha)$ and $\nabla^2 \text{Var}_{pH}(\alpha)$ are defined by (43) and (45) in Appendix A, respectively.

For Matlab code computing $\text{Var}_{pH}(\alpha)$, $\nabla \text{Var}_{pH}(\alpha)$ and $\nabla^2 \text{Var}_{pH}(\alpha)$, see the function $\text{phtv}$ in Appendix in arXiv:1602.08137 [math.NA]. For completeness, we also refer to the same preprint for corresponding formulas concerning Huber total variation.

### 2.4 Regularization with interpolating functions

As mentioned before, another way to regularize the optimization problem (8) is interpolation with so-called damage functions, which was suggested and used for 1D-structures in [8, 25, 26, 27]. The method consists in doing the FEM updating with respect to the damage parameter $\alpha$ only for indices $p$ in a subsequence $P = [P_1, P_2, \ldots, P_{n_1}]$ of $[1, 2, 3, \ldots, n]$ for an integer $n_1 \in [1, n]$, and then use interpolation for deciding the value of the remaining parameters $\alpha_p$.

For example, consider a 1D-structure that is divided into 10 groups of elements with center points $x_p$, as illustrated in Figure 2. The blue circles indicate a coarser grid of points with indices $P = [1, 4, 7, 10] = [P_1, P_2, P_3, P_4]$.

![Figure 2: $N_1(x) - N_{10}(x)$ tent functions and their piecewise linear combination.](image)

For an example of 2D-structure consider 15 damage parameters connected to the groups with center points $(x_p, y_p)$, ordered in a rectangular grid

\[
\alpha_1 \quad \alpha_4 \quad \alpha_7 \quad \alpha_{10} \quad \alpha_{13} \\
\alpha_2 \quad \alpha_5 \quad \alpha_8 \quad \alpha_{11} \quad \alpha_{14} \\
\alpha_3 \quad \alpha_6 \quad \alpha_9 \quad \alpha_{12} \quad \alpha_{15}
\]

Now the coarse grid indices are $P = [1, 3, 7, 9, 13, 15] = [P_1, \ldots, P_6]$.

For a 2D-structure, let $N_{P_l}(x_p, y_p)$ be functions with the so-called interpolation property

\[
N_{P_l}(x_p, y_p) = \delta_{l,k} \overset{\text{def}}{=} \begin{cases} 
1 & \text{if } l = k, \\
0 & \text{if } l \neq k.
\end{cases}
\]

(38a)

Then, from (38a) it follows for $\alpha(x, y) \overset{\text{def}}{=} \sum_{l=1}^{n_1} \alpha_{P_l} N_{P_l}(x, y)$ that

\[
\alpha(x_{P_l}, y_{P_l}) = \alpha_{P_l}.
\]

(38b)
The 1D case is just a particular case of these equations.

The most simple example of 1D interpolating functions are so-called tent functions. For instance, see $N_1(x) - N_{10}(x)$ plotted in Figure 2 together with their linear combination, which indicates property (38b). For a 2D-structure we have tried two simple solutions. Firstly, we have generalized the tent functions to corresponding rectangular element shape functions on a 2D-grid, similar to the suggestions in [28]. This generalization works for structures with group center points arranged in a rectangular grid. For other geometries further generalizations of this solution are needed. Secondly, we take the interpolating functions equal to triangular element shape functions as they are defined in [29, Eq. 8.32] (see Figure 3). These functions show superiority over the rectangular element interpolating functions due to much less restrictions on the points in the 2D grid on which they are defined. For our test case in Section 3 the results of these two solutions were pretty similar and only a bit better smoothing was achieved for the rectangular element interpolating functions.

Figure 3: (a) Triangular element shape functions $N_i^\Delta(x,y)$, $N_j^\Delta(x,y)$ and $N_k^\Delta(x,y)$ for triangle $\Delta = (ijk)$. (b) Their linear combination over triangle $(ijk)$.

In the case of triangular element shape functions the reduced Jacobian in (13) and (14) is found as follows. Let $L$ be the $n \times n_1$-matrix

$$L \equiv (N_{P_1} \quad N_{P_2} \quad \cdots \quad N_{P_{n_1}}) \quad \text{with} \quad N_l \equiv \begin{pmatrix} \tilde{N}_l(x_1, y_1) \\ \tilde{N}_l(x_2, y_2) \\ \vdots \\ \tilde{N}_l(x_n, y_n) \end{pmatrix} \quad \text{and} \quad \tilde{N}_l = \bigcup_{\Delta_l \text{ vertex of } \Delta_k} N_{\Delta_k}. \quad (39)$$

If $\alpha^P$ is the vector of the damage parameters in the coarse grid, then (38b) and (39) give that

$$\alpha = L \alpha^P. \quad (40)$$

Moreover, the updated $m \times n_1$ Jacobian, which now can be used in Equations (13) and (14) is

$$J_r(\alpha^P) = J_r(\alpha)L, \quad (41)$$

since by (40)

$$(J_r(\alpha^P))_{d,l} \overset{\text{def}}{=} \frac{\partial r_d}{\partial \alpha_p} = \sum_{p=1}^{n} \frac{\partial r_d}{\partial \alpha_p} \frac{\partial \alpha_p}{\partial \alpha_p} = \sum_{p=1}^{n} (J_r(\alpha))_{d,p} \frac{\partial (L \alpha^P)}{\partial \alpha_p} = \sum_{p=1}^{n} (J_r(\alpha))_{d,p} L_{p,l}. \quad (42)$$

3 Test case

We test the proposed regularization methods on a 1050 × 340 × 70 mm concrete plate, reinforced by three steel rebars of 8 mm diameter, positioned as in Figure 4. The plate was excited by a swept sine force signal using an electromagnetic shaker of type LDS V406 combined with
an amplifier LDS PA100E. Ideally, either the plate or the shaker should be freely supported 
(or grounded) [30, Section 3.3]. The plate was therefore hanging in bungee cords as shown 
in Figure 5. The input force and the corresponding driving point acceleration were measured 
by an impedance head, Brüel & Kjær 8001, each signal connected through a charge amplifier 
B&K 2635. The remaining response points were measured using accelerometers of type B&K 
4508 B002, which were attached to the structure by using mounting plastic clips B&K UA-1407 
together with a thin layer of beeswax applied inside the clips for a more firm connection. The 
accelerometers/clips were glued to the plate at 5 x 13 = 65 measurement points. A B&K 3560-C 
served as the data acquisition unit. It was controlled by a portable PC by using the software 
B&K Pulse Labshop. These measurements were done for the following five cases

Figure 4: Cross-section of the test plate (unit: mm).

Figure 5: The measurement setup.
Figure 6: (a) Linear 6.6 kN load for producing deeper cracks located at the notch. (b) Visible deeper crack after the largest applied linear load (Damage 4).

**Damage 0** Undamaged plate.

**Damage 1** A 7 mm deep notch cut with an angle grinder (Figure 7 and 9).

**Damage 2** A 13.5 mm deep notch cut with an angle grinder (Figure 7 and 9).

**Damage 3** Deeper real cracks, produced by applying a 6,6 kN linear load (Figure 6 (a)).

**Damage 4** Even deeper cracks (Figure 7), produced by using C-clamps to apply larger linear loads.

For Damage 4, the servohydraulic actuator was not available, so the plate was now again mounted as in Figure 6 (a), but the steel bar applying the linear load was pushed down with C-clamps attached to the endpoints of the steel bar and to the table under the concrete plate.

Figure 7: Visible cracks on the plate after applying larger linear loads with C-clamps. The left-most crack in (a) and the right-most crack in (b) seemed to be less deep.

By using FRF analysis, totally 12 mode shapes were identified from the measurement data, but only the first three bending mode shapes (see Figure 8 and Table 1) were used in the damage
identification. The 30 first modes were used in the finite element analysis in order to produce system matrices and compute modal data derivatives with respect to the damage parameters in (21b).

Figure 8: The first three measured bending mode shapes. Green color corresponds to the undeformed plate. (a) Mode shape no. 1: eigenfrequency $f = 249.03 \pm 0.11$ Hz. (b) Mode shape no. 3: $f = 668.40 \pm 0.52$ Hz. (c) Mode shape no. 5: $f = 1269.88 \pm 0.38$ Hz.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Undamaged plate</th>
<th>Damage 1</th>
<th>Damage 2</th>
<th>Damage 3</th>
<th>Damage 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode 1</td>
<td>249.03 ± 0.11</td>
<td>243.00 ± 0.11</td>
<td>239 ± 0.10</td>
<td>217.13 ± 0.60</td>
<td>192.60 ± 0.54</td>
</tr>
<tr>
<td>Mode 3</td>
<td>668.40 ± 0.52</td>
<td>660.99 ± 0.92</td>
<td>661.48 ± 0.22</td>
<td>638.24 ± 0.57</td>
<td>604.20 ± 0.28</td>
</tr>
<tr>
<td>Mode 5</td>
<td>1269.88 ± 0.38</td>
<td>1256.97 ± 0.80</td>
<td>1257.13 ± 0.94</td>
<td>1221.70 ± 0.58</td>
<td>1159.06 ± 0.53</td>
</tr>
</tbody>
</table>

Table 1: Eigenfrequencies (in Hertz with standard deviation) for the first 3 bending mode shapes for both undamaged and damaged cases.

The FE model of the plate is built in Abaqus software using S4R shell elements with 4 nodes each and 6 degrees of freedom at each node: 3 translational and 3 rotational. The reinforcement is modeled as a re-bar layer that is embedded in the shell section and positioned according to Figure 4. The boundary conditions are free from constraints. The plate is divided into 65 parts (groups of elements). Each part is designated with independent elasticity modulus of concrete. The model contains 1240 elements and 1323 nodes.

3.1 Summary of results

We have compared the results of FEMU for an 1D (wide beam) and 2D plate models and different regularization techniques. The 1D and 2D plate models were divided into 13 and 65 groups, respectively, as shown in Figure 9. For these group divisions when we used the total variation based regularization methods we applied 1D and isotropic 2D total variation, respectively. The results are plotted in Figures 10 to 13. The plotted parameter is the damage index $\alpha_i$ defined in Equation (5). Recall that the damages are indicated by high $\alpha_i$ values.

The elasticity modulus distribution for the undamaged plate was found by FEMU using the initial guess $E_0^i = 36.5$ GPa for all the groups. This value was obtained from the cylinder test. For all presented results, the elasticity modulus in the groups containing the two shorter edges are constrained to the range 35.9–37.2 GPa, which corresponds to maximal damage index $\alpha_i \approx 0.02$. This was suggested in [8] for avoiding unrealistic high parameter values at edges, due to lower sensitivity of modal data to changes in elasticity modulus there.

Damages 1 and 2 (the notches) are located exactly between stripes number 5 and 6 in Figure 9. For Damage 4, additional cracks were visible closer to and along the center line in stripe
Damage 1 & 2

Figure 9: Plate division into groups. The center points of the grey color rectangles correspond to the coarse mesh grid used for regularization by interpolation. (a) 13 groups. (b) 65 groups (unit: mm). Damage 1 and 2 (introduced in Section 3) are following the borderline between stripes 5 and 6, see also the location and direction of the notch in Figure 7.

Figure 10: No regularization, 13 groups, 35.9–37.2 GPa constraints for the stripes no. 1 and 13, 1–40 GPa constraints for stripes no. 2–12.

Thus the known damages are in the interval 5–8 and some smaller cracks in stripe 4 and 9.

Figures 10 and 11 show results obtained without regularization. The plotted damage indices exhibit irregular fluctuations in undamaged parts of the structure, which easily could be misinterpreted as small additional damages. This makes it quite difficult to correctly identify both the location and severity of the damage. The damage index peaks around the real damages, but there are also additional positive and negative peaks at stripes 2 and 9.

From Figure 12, we see that for the small damage, i.e. Damage 1 in our case, different regularization techniques result in almost the same damage pattern, which can be described as a bell shaped parameter distribution around the damage location at position between stripes 5 and 6. On the other hand, when the damage becomes to be more pronounced, the optimization with the pseudo Huber total variation regularization results in a more localized damage pattern compared with the results based on either the damage functions or $l_2$-norm regularization.
3.1 Summary of results

Figure 11: No regularization, 65 groups, 35.9–37.2 GPa constraints for short edges, 1–40 GPa constraints for the remaining groups.

Figure 12: Comparison of damage functions (df), $l_2$-norm and pseudo Huber total variation (phtv) regularizations, 13 groups, 35.9-37.2 GPa constraints for stripe no. 1 and 13, 1-40 GPa constraints for stripe no. 2-12.

Supports this type of damage pattern compared with more smeared bell shaped pattern.

Figure 13 shows almost the same comparison of methods as in Figure 12 but for the 2D plate.
Figure 13: Comparison of damage functions (df), $l_2$-norm and pseudo Huber total variation (phtv) regularizations, 65 groups, 35.9–37.2 GPa constraints on short edges, 1–40 GPa constraints for the remaining groups. For the damage functions the coarse mesh grid is [1 3 5 11 13 15 21 23 25 31 33 35 41 43 45 51 53 55 61 63 65].

model and 65 groups. Here there is a bigger difference between the results for the interpolation with the damage functions and those for the pseudo Huber total variation.

The pseudo Huber total variation regularization gives a sharp damage indication in stripes 5–6 for Damage 2–3, corresponding to the notch and cracks located between these stripes, whereas for Damage 4, it also gives some indication in stripes 7–9, corresponding to the additional cracks in Figure 7 (b). It gives no indication of the small cracks in Stripe 4 in Figure 7 (a), however, which could mean that those cracks are less deep than the others. The pseudo Huber penalty term also reduces small amplitude oscillations in the damage parameters. See the differences between the methods at stripes 2–4, 5–6 and 7–12 for Damage 2–4 in Figure 13. Since the pseudo Huber total variation regularization minimizes the approximation of the $l_1$-norm of the discrete gradient, the resulting solutions are expected to be nearly piecewise constant, which can be seen clearly both here and in other plots. This is a good property for finding well-localized damages.

The $l_2$-norm and the damage function regularization, on the other hand, both have the damage identification spread out over stripes 4–9 for all four damages. It is then more difficult to judge whether this indicates a spread out damage or whether it is the smoothing inherent in these methods. For a less well-localized damage, like Damage 4, all of the above methods give more similar results.

Remark 2 (Huber vs. pseudo Huber). The Huber total variation regularization might fail, for instance, if one of its second derivative discontinuities is for damage parameter values close to the solution of the minimization problem. Therefore, we have used pseudo Huber rather than Huber total variation regularization for all plots in this paper. However, it is more easy to motivate the choice of the threshold parameter $\mu$ for the Huber total variation regularization (see next section). For comparison, we have the corresponding plots for this regularization method.
with one-sided second-order derivatives in the preprint version arXiv:1602.08137 [math.NA] of this paper. The results of these two methods are quite similar both for 13 and 65 groups.

### 3.1.1 On the choice of the parameters $\mu$ and $\rho$

As a rule of thumb, the parameter $\mu$ for the Huber and the pseudo Huber function controls that any variation below this value will be smoothed out, whereas larger variations are possibly kept. In a typical application, we imagine that several measurements have been done on an undamaged structure with $\mu$ adjusted to a value that seems appropriate for smoothing small fluctuations in the the computed parameters, but also small enough for expecting a clear indication of a damage. When the first measurements are done on a damaged structure, $\mu$ can be then be adjusted further for a more clear indication of the damage, but then the previous analysis for an undamaged structure should also be computed with the same value for $\mu$, for a fair comparison of analysis results before and after the damage. For this reason, for all results in this paper, we use the same value for $\mu$ in the analysis of Damage 0–4. The chosen value for $\mu$ was found by testing and is approximately equal to the jump in the elements of the parameter vector around Damage 1 (see Section 3). Note also in Figure 14 that when the value of $\mu$ increases, the solution with the pseudo Huber total variation regularization becomes more similar to the solution with $l_2$-norm regularization as can be expected from Equation (29).

![Figure 14](image_url)

Figure 14: Comparison of the Huber total variation (htv) regularizations, 13 groups, 35.9-37.2 GPa constraints for stripe no. 1 and 13 in Figure 9 (a), 1-40 GPa constraints for stripe no. 2-12 and different values of $\mu$.

On the other hand, to find the optimal regularization parameter $\rho$ we use, when it is possible, the so-called L-curve method [31] and build a log-log-plot of the total variation norm versus the residual norm with $\rho$ as a parameter. This curve shows a trade-off between minimizing smoothing and minimizing data residual. In Figure 15 the L-curve is drawn for Damage 3 defined in Section 3 and the pseudo Huber total variation regularization with $\mu = 0.01$. In the case when only a finite number of points are known on this curve, it is popular to use a spline
approximation of this curve. Then, the optimal $\rho$ corresponds to the corner of the L-curve, which is defined as the point with maximal curvature of the cubic spline approximation. It is shown that $\rho = 0.0001$ (marked with a star) is the optimal regularization parameter. For some L-curves it was not possible to find the optimal value for $\rho$ automatically. Then the parameter $\rho$ was chosen manually around the “knee” of the L-curve. A couple of different values were tested for finding one that increases the smoothing of undesired oscillations but still keeps a sharp peak that indicates a possible damage.

More robust algorithms for finding the corner of the L-curve have been developed recently (e.g. [32, 33]), but here we just use a simple solution to find an estimate of the optimal $\rho$.

4 Conclusions

The total variation based regularization has to our knowledge not been used before for structural damage identification. We have compared this approach against the interpolation-based regularization in FEMU. Interpolation-based regularization, on the one hand, gives an automatic smoothing of the computed damage parameters at the cost of less precise localization of the damage. Regularization with (pseudo) Huber total variation penalty term, on the other hand, depends on a not fully automatic choice of parameters $\mu$ and $\rho$, but results in a more precise localization and identified severity of a well-localized damage. An inherent advantage of the (pseudo) Huber penalty term is that it also reduces small amplitude fluctuations in the damage parameters. All investigated methods give more similar results for a less well-localized damage.
5 Further work

Regularization with penalty term depends on choosing the regularization parameter ρ. The L-curve and its approximation with the cubic spline does not always give an automatic choice of the optimal ρ, so a better method for finding the optimal ρ would be desired. It would also be necessary to understand better the impact of noise on the total variation regularization methods for which a numerical finite element model could make a contribution. It could also be interesting to apply this method to a real structure, e.g. a bridge.

Acknowledgements

We gratefully acknowledge our colleges at LTU: Inge Söderkvist for a number of valuable discussions around the regularization techniques, Fredrik Ljungren for the help with vibration tests, Efthymios Koltsakis, Lennart Elfgren, Ulf Ohlsson and Zheng Huang for the discussion about the effects of damage on the structures. We also acknowledge the reviewers for the helpful suggestions which significantly improved the presentation of our results.

A Formulas related to PHTV regularization

In order to compute the gradient \( \nabla \text{Var}_{PH}(\alpha) \) and the Hessian \( \nabla^2 \text{Var}_{PH}(\alpha) \) in Equations (35) and (36), respectively, we need the first and second order derivatives of \( \varphi^P_H \) with respect to the parameters \( A_{i,j} \) and therefore \( \alpha_k \). For \( (\varphi^P_H)_{ij} = \varphi^P_H (d_{i,j}A) \), we get

\[
\frac{\partial (\varphi^P_H)_{ij}}{\partial A_{i,j}} = -\frac{1}{\mu} \frac{D_{b_{i,j}A} + D_{v_{i,j}A}}{\sqrt{1 + d_{i,j}^2/\mu^2}} \quad (42a)
\]

\[
\frac{\partial (\varphi^P_H)_{i-1,j}}{\partial A_{i,j}} = \frac{1}{\mu} \frac{D_{b_{i-1,j}A}}{\sqrt{1 + d_{i,j}^2/\mu^2}} \quad (42b)
\]

\[
\frac{\partial (\varphi^P_H)_{i,j-1}}{\partial A_{i,j}} = \frac{1}{\mu} \frac{D_{v_{i,j-1}A}}{\sqrt{1 + d_{i,j}^2/\mu^2}} \quad (42c)
\]

Then, the gradient \( \nabla \text{Var}_{PH}(\alpha) \) can be found by using Equations (42a)–(42c) as follows

\[
\nabla \text{Var}_{PH}(\alpha)_{(j-1)d_i + i} = \frac{\partial \text{Var}_{PH}(\alpha)}{\partial (\alpha_{(j-1)d_i + i})} = \left\{ \begin{array}{ll}
\frac{\partial (\varphi^P_H)_{ij}}{\partial A_{i,j}} & i = 1, j = 1 \\
\frac{\partial (\varphi^P_H)_{ij}}{\partial A_{i,j}} + \frac{\partial (\varphi^P_H)_{i-1,j}}{\partial A_{i,j}} & 2 \leq i \leq d_1, j = 1 \\
\frac{\partial (\varphi^P_H)_{ij}}{\partial A_{i,j}} + \frac{\partial (\varphi^P_H)_{i,j-1}}{\partial A_{i,j}} & i = 1, 2 \leq j \leq d_2 \\
\frac{\partial (\varphi^P_H)_{ij}}{\partial A_{i,j}} + \frac{\partial (\varphi^P_H)_{i-1,j}}{\partial A_{i,j}} + \frac{\partial (\varphi^P_H)_{i,j-1}}{\partial A_{i,j}} & 2 \leq i \leq d_1, 2 \leq j \leq d_2
\end{array} \right.
\]

The Hessian \( \nabla^2 \text{Var}_{PH}(\alpha) \) is an \( n \times n \) symmetric matrix. So one can compute its upper triangular part \( (\nabla^2 \text{Var}_{PH}(\alpha))^U \) and then expand it to the symmetric matrix. The nonzero second-order derivatives which contribute to the upper triangular part of \( \nabla^2 \text{Var}_{PH}(\alpha) \) are

\[
\frac{\partial^2 (\varphi^P_H)_{ij}}{\partial A_{i,j}^2} = \frac{1 + (D_{b_{i,j}A} - D_{v_{i,j}A})^2/\mu^2}{(1 + d_{i,j}^2/\mu^2)^{3/2}} \quad (44a)
\]
\[
\frac{\partial^2(\varphi_{PH}^{\mu})_{i-1,j}}{\partial A_{i,j}^2} = \frac{1}{\mu} \left( 1 + \frac{D_{v_{i-1,j}A}}{\mu^2} \right)^{3/2} \]

\[
\frac{\partial^2(\varphi_{PH}^{\mu})_{i,j-1}}{\partial A_{i,j}^2} = \frac{1}{\mu} \left( 1 + \frac{D_{h_{i,j-1}A}}{\mu^2} \right)^{3/2} \]

Equations (44d)–(44c) correspond to the elements on the main diagonal of the Hessian matrix. Equation (44d) is connected to the diagonal \([k, k+1]\), Equation (44e) to the diagonal \([k, k+d_1]\) and Equation (44f) to the diagonal \([k, k+d_1 - 1]\). Thus,

\[
(\nabla^2 \text{Var}_{PH}(\alpha)_{k_1,k_2})^{(\mu)} = \frac{\partial^2 \text{Var}_{PH}(\alpha)}{\partial \alpha_{k_2} \partial \alpha_{k_1}} \left\{ \begin{array}{ll}
\frac{\partial^2(\varphi_{PH}^{\mu})_{i,j}}{\partial A_{i,j}^2} + \frac{\partial^2(\varphi_{PH}^{\mu})_{i-1,j}}{\partial A_{i,j}^2} & \text{if } k_1 = k_2 = 1 \\
\frac{\partial^2(\varphi_{PH}^{\mu})_{i,j}}{\partial A_{i,j}^2} + \frac{\partial^2(\varphi_{PH}^{\mu})_{i,j-1}}{\partial A_{i,j}^2} & \text{if } k_1 = k_2 = i, 2 \leq i \leq d_1 \\
\frac{\partial^2(\varphi_{PH}^{\mu})_{i,j}}{\partial A_{i,j}^2} + \frac{\partial^2(\varphi_{PH}^{\mu})_{i,j-1}}{\partial A_{i,j}^2} & \text{if } k_1 = k_2 = (j-1)d_1 + i, 2 \leq j \leq d_2 \\
\frac{\partial^2(\varphi_{PH}^{\mu})_{i,j}}{\partial A_{i,j}^2} & \text{otherwise}
\end{array} \right. \]

References


Paper B: Modelling of Damage and its use in Assessment of a Prestressed Concrete Bridge

This Appendix consists of a reprint of the paper [HGS+16].
Modelling of Damage and its use in Assessment of a Prestressed Concrete Bridge

Zheng Huang, Yongming Tu
Southeast University, Nanjing, China and Luleå University of Technology, Luleå, Sweden

Niklas Grip, Natalia Sabourova, Niklas Bagge, Thomas Blanksvärd, Ulf Ohlsson, Lennart Elfgren
Luleå University of Technology, Luleå, Sweden

Contact: zheng.huang@ltu

Abstract

A five-span prestressed concrete bridge has been subjected to a loading test up to failure in Kiruna, Sweden. The bridge was 55 years old and had a length of 121.5 m. The test has been used to validate and calibrate existing nonlinear finite element programs for predicting the shear behavior of reinforced and prestressed concrete structures.

Two 3D finite element (FE) models of the Kiruna Bridge are built in commercial software Abaqus, one using shell-elements and one using a combination of shell and beam elements. Predictions obtained from these two models are well consistent with mode shapes and eigenfrequencies computed from acceleration measurements on the bridge before and after loading it to failure. The shear failure of the bridge is also simulated using the built-in concrete damage plasticity (CDP) model in Abaqus. The predicted load-displacement curve is in good agreement with the measurements. Verification of the CDP model is conducted at element and member level with two different damage parameter evolutions. The verification indicates that the damage parameter will affect the predicted shear behavior. It does not seem to be reliable to adopt the CDP model to simulate the shear behavior in the present research.

A long term goal is to use the measured mode shapes, eigenfrequencies and FE models for evaluating methods for damage identification. Such methods are important for maintenance of different structures, for extending their life span and for better knowledge of their load carrying capacity. The use is described of so-called sparse regularized finite element method updating (FEMU) methods. Some important properties of such methods are demonstrated using simulations on a Kirchhoff plate. For instance, the simulations suggest that both eigenfrequencies and mode shapes should be used for precise localization of the damage.

Keywords: Concrete damage plasticity model, refined shell element model, shear failure test, shear behavior, five-span prestressed concrete bridge, FEM updating, sparse regularization, Kirchhoff plate.
1 Introduction

Finite element analysis of concrete structures are now widely employed for research within structural engineering. With the development of computing technology, it is even possible to use this method to assess the existing concrete bridges which are very complex and large in size [1]-[5]. Regarding the assessment of concrete bridges, it is meaningful to combine experimental studies and finite element analysis because on one hand, the finite element model can be updated based on the measurements and on the other hand, the validated model can be used to study more detailed behavior of the bridge, which can not be obtained by the test.

Several experiments have been conducted by Luleå University of Technology on the five-span continuous prestressed bridge in Kiruna, Sweden to assess the behavior of this bridge and more importantly, to calibrate and improve the existing methods of assessment of this type of bridges [7, 8]. These experiments include: shear-failure test of the FRP (Fiber Reinforced Polymer)-strengthened girders, see Figures 1(a) and 2(a), punching shear-failure test of the bridge deck 2(b), operational modal analysis tests of the bridge before and after the failure test 2(c).

In this paper, two finite element models of the Kiruna Bridge are built. One model is based on shell-elements, which is used for simulating the shear-failure test. The other is built by combined shell and beam elements, which is aimed for damage identification of this bridge. Through modelling the test of a reinforced concrete (RC) panel and a RC beam we present a preliminary study on calibrating the damage parameter evolution of the built-in concrete damage plasticity model (CDP) in Abaqus, by which the concrete is modelled in simulating the shear-failure test.

This paper also focus on damage assessment using acceleration measurements and finite element model updating (FEMU). We investigate how certain regularization techniques can be used to give damage identification results that mimic the usually very localized (or sparse) nature of damages in real structures. We describe how such damage identification can be performed only from comparison of eigenfrequencies predicted by the FE model with eigenfrequencies measured on the real structure. Next we demonstrate on a Kirchhoff plate that one drawback with this approach is that symmetries in the structure can prevent exact assessment of the localization and severity of the damage.

The paper is organized as follows. In Section 2, we describe the Kiruna Bridge, the different tests performed on it and the developed FE models as well as the validation of the Abaqus CDP model. In Section 3, we describe some different approaches for damage identification, and, as a first step, demonstrate the use of $l_1$-norm sparse regularization on a Kirchhoff plate. We summarize our conclusions and suggestions for future research in Section 4.

2 The Kiruna Bridge

The Kiruna Bridge was a 121.5 m five-span continuous post-tensioned prestressed bridge, see Figure 1(a). The 84.2 m long western part of the bridge was curved with the radius of 500 m, while the eastern part was straight with length of 37.3 m. The bridge had a 5% inclination in longitudinal direction and 25% inclination in transverse direction. The superstructure of the bridge consisted of three post-tensioned girders which were 1923 mm in height. The bridge deck was 15.6 m wide including the edge beam. Six tendons were post-tensioned in the central and eastern segments of the bridge in each girder and four for the western segments. The profile of prestressed tendons is shown in Figure 1(c). More information about the geometry and reinforcement layout of the Kiruna Bridge can be found in [6, 8, 47, 48]. According to the design drawing, the concrete grade of the superstructure of the Kiruna Bridge was K400 ($f_{cm} = 21.4$ MPa), the steel reinforcement, denoted Ks40, had a yield stress of 410 MPa and a tensile stress of 600 MPa while the corresponding stress for prestressed tendons were 1450 MPa and 1700 MPa, respectively.
2.1 Loading to failure of the Kiruna Bridge

In order to load the girders to shear failure, the flexure capacity of the girders should be strengthened. The central girder was strengthened by attaching near surface mounted (NSM) CFRP (Carbon Fiber Reinforced Polymer) bars to the bottom while the southern girder was strengthened by CFRP laminates. The northern girder was unstrengthened. The elastic modulus and tensile strength of CFRP bars were 210 GPa and 3300 MPa, respectively. The corresponding values for the laminates were 200 GPa and 2900 MPa, respectively.

In the shear-failure test, load was applied to the mid-span of each girder in span 2-3. Firstly, all the three girders were loaded to 4 MN resulting in total 12 MN. Then, the southern girder was loaded to failure followed by loading the middle girder to failure. The failure modes of both beams were combinations of flexure and shear, including concrete crushing under the load and ultimately stirrup rupture. Extensive shear cracks can be observed at the end of the test as shown in Figure 2(a). After the test of girders, punch shear-failure test was performed on the northern part of the deck resulting in failure pattern shown in Figure 2(b). The details of the strengthening system and (b) the test results can be found in [6, 8]

2.2 3D finite element models of the Kiruna Bridge

In order to investigate if the finite element method can be employed to precisely simulate the behavior of the Kiruna Bridge, a 3D shell-element model of the Kiruna Bridge was built using the commercial FEM software Abaqus. It is well-known that shell-elements are superior to beam-elements in simulating the nonlinear shear behavior of structures. Moreover, in general, the shell-element model is computationally more efficient than the solid-element model of the same structure since smaller number of such elements is required. That is why shell-elements were used in the modelling of the Kiruna Bridge.

As it was mentioned in Section 2.1, the geometry of the Kiruna Bridge is complex (Figure 1(c)). This affects the model eigenfrequencies and mode shapes.
The ordinary steel reinforcement bars were modelled by smeared re-bars embedded in the shell section and the curved prestressed tendons were simulated by truss-element. As adjustment devices have been installed at the bottom of all columns which enable rotation of the column-end as shown in Figure 1(b), only the three translation degrees of freedom at the column-end are constrained while the rotational ones kept free. In order to assess various behavior of this bridge (e.g. the static behavior subjected to various loads and actions, the dynamic behavior subjected to moving load), the shell-element model was built consistent with the design drawing of the bridge both in geometry and in reinforcement layout, see Figure 3(a).

Figure 2: (a) Shear failure test of the girders. (b) Punching shear failure of the deck, (c) Operational modal analysis test.

Figure 3: (a) Geometry of the 3D shell element model. (b) Geometry of the shell-beam element model. (c) Modelling of the prestressed tendons in shell-element model.

The predicted modal data, i.e. eigenfrequencies and mode shapes, of the shell-element model are in good agreement with the measurements as shown in Table 1. This justifies that the stiffness and mass of the bridge are modelled consistently with the real bridge. However, regarding the research on damage
identification based on the modal data, the size of shell-element model is still too large (62413 nodes) from the computational point of view. In order to overcome this drawback, a shell-beam element model was built without much compromise of accuracy. It has 4125 nodes which is only 7% of that of shell-element model and the corresponding predicted results also show good agreement with the measurement, see Table 1.

Table 1. Eigenfrequencies $f_{\text{shell}}$ and $f_{\text{shell-beam}}$ for the FE models described in Section 2.2. Comparison of eigenfrequencies $f$ and damping ratio $\xi$ on the undamaged bridge against the same modal data on the damaged bridge and the frequencies predicted by the FE models. Notation: "value ± standard deviation" and $\Delta f = \frac{f_{\text{undamaged}} - f_{\text{damaged}}}{f_{\text{undamaged}}} \times 100$ % denotes the relative change of the eigenfrequencies after loading the bridge to failure.

<table>
<thead>
<tr>
<th>Mode nr</th>
<th>$f_{\text{shell}}$ (Hz)</th>
<th>$f_{\text{shell-beam}}$ (Hz)</th>
<th>$f_{\text{undamaged}}$ (Hz)</th>
<th>$f_{\text{damaged}}$ (Hz)</th>
<th>$\Delta f$ (%)</th>
<th>$\xi_{\text{undamaged}}$ (%)</th>
<th>$\xi_{\text{damaged}}$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8172</td>
<td>1.8022</td>
<td>1.61 ± 0.005</td>
<td>1.951 ± 0.009</td>
<td>-3.1</td>
<td>2.97 ± 0.492</td>
<td>0.91 ± 0.12</td>
</tr>
<tr>
<td>2</td>
<td>2.0397</td>
<td>2.0397</td>
<td>4.037 ± 0.009</td>
<td>4.037 ± 0.009</td>
<td>-0.0</td>
<td>1.88 ± 0.14</td>
<td>0.85 ± 0.01</td>
</tr>
<tr>
<td>4</td>
<td>4.7011</td>
<td>4.7011</td>
<td>5.032 ± 0.002</td>
<td>4.923 ± 0.002</td>
<td>-10.0</td>
<td>8.513 ± 0.014</td>
<td>3.63 ± 0.006</td>
</tr>
<tr>
<td>7</td>
<td>5.0348</td>
<td>5.0878</td>
<td>5.926 ± 0.001</td>
<td>5.926 ± 0.001</td>
<td>-4.8</td>
<td>1.89 ± 0.16</td>
<td>2.44 ± 0.000</td>
</tr>
<tr>
<td>8</td>
<td>—</td>
<td>—</td>
<td>5.926 ± 0.001</td>
<td>5.926 ± 0.001</td>
<td>-4.8</td>
<td>3.727 ± 0.016</td>
<td>1.39 ± 0.030</td>
</tr>
<tr>
<td>9</td>
<td>5.7191</td>
<td>5.7997</td>
<td>6.067 ± 0.005</td>
<td>5.976 ± 0.005</td>
<td>-3.3</td>
<td>2.87 ± 0.006</td>
<td>1.87 ± 0.000</td>
</tr>
<tr>
<td>10</td>
<td>6.6574</td>
<td>6.7016</td>
<td>7.692 ± 0.001</td>
<td>6.901 ± 0.002</td>
<td>-10.1</td>
<td>2.00 ± 0.17</td>
<td>2.97 ± 0.005</td>
</tr>
<tr>
<td>12</td>
<td>8.0929</td>
<td>8.1570</td>
<td>8.552 ± 0.000</td>
<td>7.876 ± 0.000</td>
<td>-11.4</td>
<td>1.66 ± 0.000</td>
<td>3.87 ± 0.000</td>
</tr>
<tr>
<td>16</td>
<td>—</td>
<td>—</td>
<td>19.447 ± 0.005</td>
<td>13.99 ± 1.005</td>
<td>-29.0</td>
<td>1.69 ± 0.54</td>
<td>0.54 ± 0.336</td>
</tr>
</tbody>
</table>

The shell-element model was also used to simulate the shear-failure test of the Kiruna Bridge. The concrete material was modelled by built-in Abaqus concrete damage plasticity (CDP) model and the steel was modelled by isotropic plasticity model. The law of damage parameter evolution was defined according to that presented in [4, 5]. It should be noted that the material parameters were defined according to the design drawing not the test and the three girders were loaded with equal displacement in the FEM model which is different from the test. Even if there are discrepancies between the model and the real test, consistent predictions of the load-displacement behaviour of this bridge can be obtained as shown in Figure 4. However, based on these results we can not conclude that the considered Abaqus CDP model is suitable for simulation of the shear behavior of concrete structures which will be discussed in the following section.
2.3 Verification of the Abaqus CDP model

The CDP model, which was probably first introduced in [9], is now used in Abaqus in the following formulation

\[ \sigma = (1 - D)E_1^\text{E}(\varepsilon - \varepsilon^p) = E_1(\varepsilon - \varepsilon^p), \]

where \( \sigma \) is a Cauchy stress tensor, \( D \) is a stiffness degradation, also called, damage parameter, \( E_1 \) - elastic stiffness tensor of the undamaged material, \( E_1 = (1 - D)E_1^\text{D} \) elastic stiffness tensor of damaged material, \( \varepsilon \) - strain tensor, \( \varepsilon^p \) - plastic strain tensor and \( \cdot \cdot \cdot \) double-dot product. In Abaqus this formulation requires

- stress-strain relations \( \sigma_1(\varepsilon_1^{\text{E}}) \) and \( \sigma_2(\varepsilon_2^{\text{E}}) \) for the uniaxial material behaviors under compressive and tensile loadings, where \( \varepsilon_1^{\text{E}} \) is compressive inelastic strain and \( \varepsilon_2^{\text{E}} \) is tensile cracking strain.
- damage parameter evolution \( D = D(d_1, d_2) \) described by two independent uniaxial degradation variables \( d_1 \) and \( d_2 \) under compressive and tensile loadings, respectively.

The former come from the material test and the later are usually found by "trial and error".

Figure 4: (a) Load-displacement curve of the Kiruna Bridge. (b) Shear stress-strain curve of RC panel B1. (c) Load-displacement curve of RC beam OA1.

To verify the reliability of different material models a large number of reinforced concrete panels have been tested in University of Toronto [10, 11] and University of Houston [12]. In these tests, the RC (reinforced concrete) panels were designed with a large variety of concrete grades and reinforcement ratios and subjected to various of combination of evenly distributed compression, tension and shear until failure. This experimental data became benchmark tests and has been widely used to calibrate and validate the material model at the element level [13-15]. The accepted guideline now according to FIB proposition [16] is that in order to produce reliable simulation of the behavior of concrete structures, the material models of a commercial software need to be calibrated and validated using element level benchmark tests and member level benchmark tests in advance.

2.3.1 Damage parameter evaluation

For the validation of the CDP model in this paper we use the following two models of the damage parameter evolution in terms of the evolution of the degradation variables \( d_1 \) and \( d_2 \).

In the first, \( d_1 \) and \( d_2 \) were initially proposed in [17, Table 2] and then adjusted in [4, Table 2]. The latter authors used the defined parameters to make the predicted behavior of the Övik Bridge closer to the measurements. We should note here, that the degradation variables evolutions suggested in [4] were also used for the prediction.
of the Kiruna Bridge behavior as mentioned in the previous section.

Secondly, we use the degradation variable evolutions as functions of the compressive damage parameter \( b_2 \) and tensile damage parameter \( b_2 \) suggested in [18] and utilize here the following slightly modified formulas from [18]

\[
d_s \left( b_2, \sigma_{\text{cr}}^{\text{f}}, E_0 \right) = \frac{(1 - b_2)\varepsilon_0^{\text{f}}}{(1 - b_2)\varepsilon_0^{\text{f}} + \varepsilon_0^{\text{f}} - \varepsilon_0^{\text{f}}} - 1
\]

\[
d_t \left( b_2, \sigma_{\text{cr}}^{\text{f}}, E_0 \right) = \frac{(1 - b_2)\varepsilon_0^{\text{f}}}{(1 - b_2)\varepsilon_0^{\text{f}} + \varepsilon_0^{\text{f}} - \varepsilon_0^{\text{f}}} - 1
\]

Moreover, \( b_2 = \frac{\varepsilon_0^{\text{f}}}{\varepsilon_0^{\text{f}}} \) and \( b_2 = \frac{\varepsilon_0^{\text{f}}}{\varepsilon_0^{\text{f}}} \) where \( \varepsilon_0^{\text{f}} \) is the compressive plastic strain and \( \varepsilon_0^{\text{f}} \) is the tensile plastic strain, while \( \varepsilon_0^{\text{f}} \) and \( \varepsilon_0^{\text{f}} \) were explained previously. The parameters \( b_2 \) and \( b_2 \) can vary from 0 to 1, where 1 means no damage and 0 means total damage. In what it follows we set \( b_2 = b_2 = 0.9 \) according to the calibration at the element level, see Section 2.3.2

2.3.3 Model evaluation at the element level

In this paper we verify the CDP model using the RC panel named B1 defined in [19]. The panel is reinforced by orthogonal steel re-bars with reinforcement ratio of 0.19% in the longitudinal and transverse direction, respectively. This panel was subjected to pure shear to failure.

Figure 4(c) shows the comparison of the shear stress-shear strain curve between the simulation and the experiment, where the model CDP-1 is based on the damage parameter \( D(a_2^2, a_2^2) \) and CDP-2 is connected to \( D(a_2^2, a_2^2) \). Clearly, CDP-1 model cannot reflect the shear stiffening effect of reinforced concrete due to tension stiffening and aggregates interlock after diagonal cracks emerge. Once cracking occurs, the shear stiffness declines sharply and the predicted stress-strain curve is inconsistent with the experimental result. On the other hand, the CDP-2 model produces more consistent predicted results. It seems that the damage parameter will affect the shear behavior of reinforced concrete which hasn’t been mentioned by other researchers before.

2.3.3 Model evaluation at the member level

At the member level, one RC beam without shear reinforcement tested in [20], denoted OA1, is simulated using model CDP-1 and CDP-2 which are described in the previous section. The details of the beam can be found in [20]. All material parameters of concrete are derived from cylinder compressive strength using the expression proposed by the fib Model Code [21].

In the first simulation, CDP-1 is employed and the comparison of the prediction and experimental results is shown by the red line in Figure 4(c). Good agreement between the prediction and test can be found in terms of the load-displacement curve. Regarding the crack pattern at peak load, the experiment indicates an inclined crack initiating in the shear span and propagating to the top of the beam causing the final failure as shown in Figure 5(a). However, the predicted result presents a flexure crack initiating near the mid-span of the beam causing the final failure as shown in Figure 5(b) which indicates this model can’t simulate the shear cracking behavior of the beam.

Figure 5: Crack pattern at peak-load of RC beam OA1. (a) Experiment. (b) CDP-1. (c) CDP-2.

Why will these two contradictory conclusions be reached when it comes to simulating this beam?
The authors in [22] simulated the behavior of the same beam based on Euler theory which can only take the flexure deformation into account while neglecting the shear deformation. It is shown that even with this assumption the predicted load-displacement curve is still consistent with the experimental results. It implies most of the deflection at the mid-span is caused by flexure deformation while the shear deformation can be neglected in this case. Namely, CDP-1 model can describe the flexure behavior of reinforced concrete accurately but faila to model the shear behavior (shear cracking).

In the second simulation, CDP-2 is adopted and a good agreement between the prediction and measurement can be found regarding the crack pattern as shown in Figure 5(a) and Figure 5(c). However, the predicted load-displacement curve overestimates the peak load which is illustrated as the yellow line in 4(c).

2.4 Acceleration measurements and modal analysis results

Accelerometer measurements of ambient vibrations were performed in May 2014 on the undamaged bridge and twice in August 2014 on the damaged bridge.

Measurements were done with six calibrated [23, 24] Colibrys SF3000L triaxial accelerometers connected with 40–60 m long twisted pair cables to an MGC-Plus data acquisition system using AP801 cards with sample rate 800 Hz. The accelerometers were firmly attached to the bridge with expansion bolts and adjusted to the horizontal plane with three screws. Figure 2 (c) shows the 38 accelerometer locations on the bridge.

Nonlinear trending in the signals was reduced by a smooth padding of the measurements (to reduce discontinuities in the periodized signal) followed by highpass filtering. Measurements that were distorted by malfunctioning electrical power supply were excluded from the analysis.

To reduce problems with low signal-to-noise ratio due to nearly no excitation from wind or traffic, we did several hours long measurements and for the damaged bridge, also tried combining two measurement occasions.

Operational modal analysis with all methods available in the software ARTeMIS 4.0 for different combinations of measurement data gave the the eigenfrequencies ($\omega$) and damping ratios ($\zeta$) that are summarized in Table 1. We have there restricted to modes with small frequency standard deviation and realistic damping ratio that were found both in the May and August measurements. See [25] for details. The measured eigenfrequencies are lower for the damaged bridge, which also is what to expect from damage theory. For the undamaged bridge, the measured mode shapes computed by ARTeMIS are plotted in Figure 6.

![Figure 6. Selected mode shapes for the undamaged bridge. Plotted mode shapes in [25] show that the predicted and measured mode shapes are quite similar for vibration modes 1, 2, 10 and 12 in Table 1. These are the vibration modes that seem most useful for damage identification.](image)

3 Structural damage identification using FEMU

There exist a lot of methods used for structural damage identification [26 -28]. One of the most computationally efficient and recognized is damage detection using sensitivity-based finite element model updating. The finite element model is then initially parameterized by unknown parameters, which are iteratively updated by some parameter estimation method, usually nonlinear least squares. When the uncertain
parameters are updated the derivatives and sometimes even second-order derivatives of the modal data with respect to these parameters are used [29-33]. The corresponding matrices are often called sensitivity matrices, which is reflected in the method name. Recently it became more and more popular to use formulation of the sensitivity-based damage identification as a convex problem [34, 35] for which there exist special efficient optimization algorithms [36]. Furthermore the researchers recognized that damage is a rather local phenomena and started to use sparse regularization in order to reflect this phenomena [34, 35, 37, 38]. There exist a number of free open-source Matlab optimization packages that offer all necessary tools to solve such convex sparse regularized problems [39, 40].

Our goal is to apply sparsity together with convexity for the damage identification of the Kiruna Bridge using a SHM finite element model updating package that is developed at Luleå University of Technology and described in more detail in [33]. Most attempts in this direction are applied to simulated data. We also decided to first develop a finite element model of a Kirchhoff plate and investigate the limitations and advantages of these techniques on simulated data, which is the topic of this section.

3.1 Damage parametrization

A discrete linear time-invariant model of structural motion which is used in damage identification process is described by a second-order differential equation:

\[ M\ddot{u}(t) + C\dot{u}(t) + K u(t) = f(t), \]  

(2)

where the matrices \( M, C, \) and \( K \) are real time-independent square system mass, damping and stiffness matrices of order \( d \times d \) corresponding to the number of degrees of freedom of the model and \( u(t) \) is a time dependent displacement vector with \( d \) entries. Dots represent derivatives with respect to time \( t \) and \( f(t) \) is a vector of external forces. Considering the free vibration of an undamped structure, i.e. \( f(t) = 0 \) and \( C = 0 \) and looking for the harmonic solution of Equation (2) in the form \( u(t) = e^{j\omega t} \), we obtain the following generalized eigenvalue problem

\[ (-\lambda_k M + K)\phi_k = 0. \]  

(3)

Here, \( \lambda_k = \omega_k^2 = (2\pi f_k)^2 \) and \( \phi_k \) are the \( k^{th} \) eigenvalue and mass-normalized eigenvector, respectively, whereas \( f_k \) is the \( k^{th} \) eigenfrequency. From Equation (3) it is easy to see that changes in system matrices \( M \) and \( K \) cause changes in the modal parameters \( \lambda_k \) and \( \phi_k \).

It is very popular to assume that the mass of the undamped structure does not change after the damage is introduced and to update the stiffness matrix by the substructure matrices [29 - 31] as follows

\[ K(\alpha) = K^0 - \sum_{i=1}^{I} \alpha_i K_i. \]  

(4)

Here \( K(\alpha) \) is the improved stiffness matrix of the parameterized model. \( K_i \) is the constant expanded order matrix for the \( i^{th} \) element or substructure (group) representing the unknown model property and location. The widely used dimensionless updating or damage parameters are chosen according to the simple isotropic damage theory [41]. In this theory, the damage is described by a reduction in bending stiffness, measured as follows

\[ \alpha_i = \frac{E_i^0 - E_i}{E_i^0}. \]  

(5)

where \( E_i^0 \) and \( E_i \) is the initial (undamaged) and updated (damaged) elasticity modulus, respectively. The matrix \( K^0 \) in (4) is then interpreted as the matrix corresponding to the undamaged structure.

Thus, the parameterized eigenvalue problem reads as

\[ K(\alpha) \phi_k(\alpha) = \lambda_k(\alpha)M \phi_k(\alpha), \]  

(6)

Clearly, a small value of \( \alpha_k \) or zero in the ideal case, indicates the absence of damage for a
particular element or group, positive $\alpha_i$ corresponds to decrease and a negative $\alpha_i$ indicates increase of the elasticity modulus for the element or group.

3.2 Convex formulation of the optimization problem

In order to solve the parameter estimation problem, we measure the difference between the measured and analytical properties with an eigenvalue residual

$$\mathbf{r}(\alpha) = \omega_j \left( 1 - \frac{\lambda_j(\alpha)}{\lambda_j^{\text{finite}}} \right), \quad j = 1, \ldots, m, \quad (8)$$

where $\omega_j$ is the $j$th component of the weighting vector, $\lambda_j^{\text{finite}}$ is the $j$th measured eigenvalue, $m$ is the number of compared eigenfrequencies and $\alpha$ is the damage parameter vector, see equation (5).

To find the damage parameters we minimize $1/2 \| \mathbf{r}(\alpha) \|_2^2$. This problem is solved in each iteration by linearizing the residual as follows

$$\mathbf{r}(\alpha^{(n+1)}) = \mathbf{S}(\alpha) \Delta \alpha^{(n)} + \mathbf{r}(\alpha),$$

which is valid for a small variation $\Delta \alpha^{(n)}$ around $\alpha^{(n)}$. The matrix $\mathbf{S}$ for the partial derivatives of residuals with respect to the updating parameters is also known as the sensitivity matrix and defined as

$$S_{ij} = \frac{\partial (\mathbf{r}_j)}{\partial \alpha_i^{(n)}} = -\omega_j \lambda_i^{(n)} \frac{\partial \lambda_j^{(n)}}{\partial \alpha_i^{(n)}}.$$  

For non-repeated eigenvalues and corresponding mass-normalized eigenvectors, the following Fox-Kapoor formula [42] together with the parametrization (6) gives

$$\frac{\partial \lambda_j^{(n)}}{\partial \alpha_i} = \phi_j^{(n)^T} K_i^{(n)} \phi_j^{(n)} = -\phi_j^{(n)^T} K_i^{(n)} \phi_j^{(n)}.$$  

For repeated Eigenvalues, we refer to [43]. The resulting minimization problem is

$$\min_{\Delta \alpha^{(n)} \in \Omega} \frac{1}{2} \| \mathbf{S}(\alpha^{(n)}) \Delta \alpha^{(n)} + \mathbf{r}(\alpha^{(n)}) \|_2^2, \quad (9)$$

Then, in each iteration step the updating parameter vector is updated as $\alpha^{(n+1)} = \alpha^{(n)} + \Delta \alpha^{(n)}$ (see also [44, Eq. 19]).

The minimization problem (9) is a convex problem. Namely, the set $\Omega = \{ \Delta \alpha^{(n)}, \mathbf{S}(\alpha^{(n)}) \Delta \alpha^{(n)} = -\mathbf{r}(\alpha^{(n)}) \}$ is convex. In fact, at each iteration step the matrix $\mathbf{S}(\alpha)$ is defined at the previous step and thus it is considered as being constant. Thus, for any $t \in [0, 1]$ and $\Delta \alpha^{(n)}_1, \Delta \alpha^{(n)}_2 \in \Omega$ we have

$$\mathbf{S}(\alpha)(t \Delta \alpha^{(n)}_1 + (1-t) \Delta \alpha^{(n)}_2) =
\begin{align*}
t \mathbf{S}(\alpha) \Delta \alpha^{(n)}_1 + (1-t) \mathbf{S}(\alpha) \Delta \alpha^{(n)}_2 =
\end{align*}$$

$$= t \mathbf{r}(\alpha) - (1-t) \mathbf{r}(\alpha) = -\mathbf{r}(\alpha)$$

and therefore $t \Delta \alpha^{(n)}_1 + (1-t) \Delta \alpha^{(n)}_2 \in \Omega$.

3.3 Problem regularization

In the presence of noise in the measured observations, the estimated parameters found by an iterative method (9) can have a pronounced tendency to form an oscillating pattern that makes it difficult to localize and quantify the damage [33, Figures 12 and 13]. A standard solution of this problem is to use a regularization technique

$$\min_{\Delta \alpha^{(n)} \in \Omega} \frac{1}{2} \| \mathbf{S}(\alpha) \Delta \alpha^{(n)} + \mathbf{r}(\alpha) \|_2^2 + \beta R(\Delta \alpha^{(n)}), \quad (10)$$

where $\beta$ and $R$ are the regularization parameter and the regularization function, respectively. The regularization function describes the properties of the expected solution, for example, measure of smoothness, sparsity, etc. Below we describe two regularization methods.

3.3.1 $l_2$-norm or Tikhonov regularization

Tikhonov or $l_2$-norm regularization belongs to traditional and most used regularization method [30, 44]. It smooths the solution significantly and thus results in the solution vector full of nonzero elements [36].
3.3.2 Sparse regularization with $l_1$-norm

The nature of the damage is quite local and sometimes is compared with mathematical $\delta$ function. So the damage is associated only with few locations on a structure and thus the damaged elements are sparse compared to all the elements used in the model of the structure. The most simple and intuitive measure of sparsity of vector $x$ as a solution of the underdetermined system of linear equations $Ax = b$, where $A \in \mathbb{R}^{m \times n}$ for $m < n$, is by counting the number of nonzero entries in it or using, so-called, $l_0$ "norm"

$$\|x\|_0 = \#\{i : x_i \neq 0\}.$$  

It is not really a norm, since it does not satisfies the homogeneity property.

The $l_1$-norm regularization problem belongs to the class of combinatorial problems, which are computationally difficult [45]. That is why for simplicity its closest convex relaxation $l_1$-norm is used in regularization instead

$$\min_{\Delta \alpha^{(0)}} \frac{1}{2} \| S^{(0)} \Delta \alpha^{(0)} + \Delta \alpha^{(1)} \|_2^2 + \beta \| \Delta \alpha^{(0)} \|_1,$$  

(11)

where $\| \Delta \alpha^{(0)} \|_1$ is often called sparsifying term. Regularization with $l_1$-norm leads to sparse solution with only few nonzero elements [36].

There are different technical sufficient conditions under which the solution of the $l_1$-norm regularization coincide with the solution of the $l_0$-norm regularization, and thus (for the right choice of $\beta$) is guaranteed to be optimally sparse. See, for instance, [46], for a lengthy discussion and further references. We will see examples of sparse but not optimally sparse solutions in next section, as discussed in Section 3.4.2.

3.4 Simulation results for a damage on a Kirchhoff plate

3.4.1 Kirchhoff plate

We test the regularization methods on a square plate with size $1 \times 1 \times 0.01$ m (c.f. [35]). The initial elastic modulus for all elements is set to 200 GPa. The model is built using shell elements with 4 nodes each and 6 degrees of freedom: three translational and 3 rotational. The size of each finite element is $0.05 \times 0.05$ m, thus the model contains 400 elements. The plate is fixed on all sides. The elements are numbered as follows

The plate is built as assembly of parts which is tested in the framework of the SHM finite element model updating package [25] and cvx open-source code [39].

3.4.2 Simulation results

Figure 7 (a) shows a damage at element 211. For symmetry reasons, this damage gives exactly the same vibration mode eigenfrequencies and residual vector as an identical damage in element 210, 190 or 191. In fact, from a physical point of view, it is just the same plate with the same boundary conditions rotated 90, 180 or 270 degrees. Thus there is no way for a damage identification method to tell these four damages apart only from a comparison of eigenfrequencies.

At best, if the $l_1$-norm regularization (12) gives an optimally sparse solution with only one nonzero element, it will indicate a damage in one of the elements 190, 191, 210 and 211 (with 25 % chance of picking the right one). In Figure 7 (b)–(d), we see that as the number of eigenfrequencies used in the residual vector increase from 3 to 10, the location of the indicated damage is narrowed down from four to two of the elements 190, 191, 210 and 211.
Moreover, the computed sums of $\alpha_i$s are very close to the 10 % stiffness reduction in the damaged element, so the severity of the damage is correctly estimated, but it is distributed on a larger number of elements. Roughly equal to the sum of stiffness reductions (10 %) of the six damaged elements, but now distributed on a larger number of elements. In Figure 8 (e), finally, we see that $l_2$-norm regularization again gives a less sparse and more smooth solution.

**Figure 7**: Damage 10% at element no. 211 with reduction in elasticity modulus from 200 GPa to 180 GPa. No noise is added. Boundary conditions for $\Delta\alpha = 0$ to 80%. $l_1$-norm regularization with $\beta = 0.000001$ and identity pairing of eigenfrequencies. (a) True damage location. (b) First 3 eigenfrequencies. $\alpha(211) = \alpha(190) = \alpha(210) = \alpha(191) \approx 2.11\%$. $\Sigma_{\Delta\alpha=0.01} \alpha_i \approx 8.93\%$ (4 elements). (c) First 7 eigenfrequencies. $\alpha(211) = \alpha(190) \approx 3.99\%$, $\alpha(210) = \alpha(191) \approx 1.39\%$. $\Sigma_{\Delta\alpha=0.01} \alpha_i \approx 10.02\%$ (4 elements). (d) First 10 eigenfrequencies. $\alpha(211) = \alpha(190) \approx 4.85\%$, $\alpha(210) = \alpha(191) \approx 0.17\%$. $\Sigma_{\Delta\alpha=0.01} \alpha_i \approx 10.04\%$ (2 elements).

Figure 8 (a) and (b) shows the corresponding results for $l_2$-norm regularization. As expected, we see that it gives a more smoothed and less sparse solution than the $l_1$-norm regularization. Figure 8 (c) shows a damage that resembles two parallel cracks. In (d), we see that $l_1$-norm regularization gives damage identification with very roughly the right localization, as well as a symmetrically placed “ghost damage” for the same reasons as explained above. Moreover, the computed sum of $\alpha_i$s still is roughly equal to the sum of stiffness reductions (10 %) of the six damaged elements, but now distributed on a larger number of elements. In Figure 8 (e), finally, we see that $l_2$-norm regularization again gives a less sparse and more smooth solution.

**Figure 8**: (a)-(b): Identical settings as in Figure 7 except for using $l_2$-norm regularization. (a) First 3 eigenfrequencies $\alpha(211) = \alpha(190) = \alpha(210) = \alpha(191) \approx 1.61\%$. $\Sigma_{\Delta\alpha=0.01} \alpha_i \approx 6.42\%$ (4 elements). (b) First 10 eigenfrequencies. $\alpha(211) = \alpha(190) \approx 2.04\%$, $\alpha(210) = \alpha(191) \approx 2.24\%$. $\Sigma_{\Delta\alpha=0.01} \alpha_i \approx 8.85\%$ (4 elements). (c)-(e): Two parallel cracks with 10% stiffness reduction at each damaged element. The same $\beta$ and pairing as in Figure 7 (c) True damage location in elements no. 54, 75, 92, 96, 113, 134. (d) $l_1$-norm regularization, first 20 eigenfrequencies. $\Sigma_{\Delta\alpha=0.01} \alpha_i \approx 55.81\%$. (e) $l_2$-norm regularization, first 20 eigenfrequencies. $\Sigma_{\Delta\alpha=0.01} \alpha_i \approx 60.45\%$. 


4 Conclusions and future work

In conclusion, in order to verify the capability of a material model to simulate the shear behavior of reinforced concrete structures, not only the load-displacement curve but also the local reaction such as strain distribution and crack pattern should be compared with the member level benchmark test. Based on the present research, it is not reliable to adopt the CDP model to simulate the shear behavior of reinforced concrete structures because consistent predictions of both load-displacement curve and crack pattern compared to the measurement cannot be obtained. Further research on calibrating the damage parameter evolution of this model should be performed.

In our test of damage identification using $\ell_1$-norm regularization on the Kirchhoff plate, we got more sparse solution than with $\ell_2$-norm regularization, but still not optimally sparse. For an optimally sparse solution, we suggest to extend the residual to also contain a comparison of predicted and measured mode shapes. Then a next step can be to try applying the same sparse regularization on larger and more complicated structures, such as the Kiruna Bridge. We explained shortly in Section 2.4 how measurements and modal analysis on that bridge were performed before and after loading that bridge to failure, and found at least four mode shapes suitable for the damage identification.

5 Acknowledgements

The authors were supported by grants from the Swedish Research Council Formas (registration numbers 2007(1430) and 2012(1037) as well as from the Swedish Construction Industry's Organisation for Research and Development (SBUF), grant 13010.

Zheng Huang was supported by the National Natural Science Foundation of China (project number 51378104).

6 References


Referenser


[SGT+17] Natalia Sabourova, Niklas Grip, Yongming Tu, Chao Wang, Ola Enochsson, Thomas Blanksvärd, Martin Nilsson, Ulf Ohlsson, and Lennart Elfgren. Railway concrete arch bridge over Kalix river at Långforsen — dynamic properties and load-carrying capacity. Research report, Division of Structural Engineering, Department of Civil, Environmental and Natural Resources Engineering, Luleå University of Technology, SE-971 87 Luleå, Sweden, [SGT+17].


Sakregister

The list of symbols is sorted alphabetically by the names of the defined symbols. For example, all kinds of brackets are sorted under “b”, $A$ under “s” (as in star), $\overline{a}$ under “o” (as in overline) and $\tilde{q}$ under “t” (as in tilde). **Bold** page numbers are used to indicate pages with important information about the entry, e.g., a precise definition or the most detailed explanation found in this thesis, while page numbers in normal type indicate a textual reference.

- analysis mapping, 113
- canonical dual frame, 113
- ?, 237
- AbaqusModalDataSim, 118
- AbaqusPkg, 118
- AbaqusUtilities, 121
- ArtemisPkg, 153
- AverageWeighting, 167
- BoundConstraints, 196
- Constraints, 197
- DerRepeatedEigenvalues, 167
- DiffUniformWeighting, 171
- ElastModOptProblem, 197
- FemOptParamConvMethod, 197
- FoxKapoor, 172
- FreqResidual, 173
- GenNelsonRepeatedEigenvalues, 174
- GenNelsonRepeatedEigenvaluesNew, 174
- GenNelsonRepeatedEigenvalues, 170
- GeneralSection, 142
- GeneralUtilities, 216
- Geometry, 241
- GradientData, 181
- GradientMethod, 182
- IBFemOptParamConv, 198
- IdentityPairing, 182
- L2NormResidual, 183
- LineSegmentsIntersect, 239
- MACPairing, 183
- MACResidual, 186
- MSFResidual, 190
- MSFScaling, 191
- MZCPlate, 160
- Material, 145
- MatlabModalDataSim, 158
- MatlabSimPkg, 158
- ModalDataPkg, 167
- ModalData, 187
- ModePairingMethod, 190
- ModeScalingMethod, 190
- NTROptimizer, 206
- ObjectiveFunction, 208
- OptProblem, 210
- OptimPkg, 196
- Optimizer, 210
- OutFunClass, 211
- PointwiseWeighting, 192
- ReadGroupDefFile, 241
- ReferenceBasedResidual, 192
- ResidMethod, 193
- ResultType, 213
- SignedArea, 238, 239
- SimpleResidual, 194
- SimulatorPkg, 213
- Simulator, 213
- SpringElemNodeData, 149
- Spring, 147
- Structure, 166
- UniformFreqWeighting, 195
- UniformShapeWeighting, 195
- UniformWeighting, 195
- UpdElastElement, 153
- UpdatingStructure, 150
- Utilities, 153
- UtilityPkg, 214
- WeightingMethod, 196
- convertFem2OptCon, 198
- convertFem2Opt, 198
- convertOpt2Fem, 198
- delete, 214
- disconnect, 214
- disp, 214
dlnode, 214
evaluate, 208
getModeNrs, 187
getNu, 187
getPhi, 187
grPlemNodeNrs, 117
insertAfter, 214
insertBefore, 214
interpolFunctions, 199
lineSegmentsIntersect, 238
makeIndexVectors, 239, 240
nodeNrCoords, 117
pairModes, 184
polyg2Clockwise, 239
setModeNrs, 187
setNu, 187
setPhi, 187
signedArea, 237
sortRectMeshPts, 199
zCoordsForPlane, 199

45° Cracks Dictionary, 116
basis, 113
C (damping matrix), 96
c<br>o</br>v</br>e</br>x</br> convex problem, 112

D (total number of DOFs), 96
D0 (number of DOFs per node), 96
Degree of Freedom (DOF), 96
degree of freedom (DOF), 96
DFT (Discrete Fourier Transform), 64
DOF (Degree of Freedom), 96
DOF (degree of freedom), 96

f(a) (objective function), 99
FEMU (FEM updating), 38
frame, 113, 116
generalized eigenvalue problem, 96
Haar Dictionary, 116
interpolation property, 105
Jacobian, 99
K (stiffness matrix), 96
Kronecker Delta Dictionary, 115
M (number of modes), 96
M (mass matrix), 96
Mercedes–Benz frame, 114
modal scale factor (MSF), 98, 98
mode shape, 97
MSF (modal scale factor), 98, 98
νa (analytical predicted mode frequency), 96
νm (eigenfrequency), 97
νm (measured mode frequency), 96
objective function, 38, 99
objective function (f(a)), 99
P (number of updating parameters), 96
φa (analytical predicted mode shape), 96, 97
φm (measured mode shape), 96, 97
r(a) (residual), 99
residual (r(a)), 99
sensitivity matrix, 112
signed area, 237
tent function, 105