Escola Tècnica Superior d'Enginyeries
Industrial i Aeronàutica de Terrassa
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Department de Resistencia de Materials

Grau en Enginyeria en Tecnologies Aeroespacials

ANNEX
Study of the model-order reduction of the aerolastic behavior of a wing

Final Degree Thesis of:
Rodeja Ferrer, Pep

Director:
Prof. Joaquin Hernández Ortega

Co-director:
Prof. Juan Carlos Cante

June 2016

Contents

1 Data Preparation

1.1 Data extraction

The data needed to perform the MOR is the stiffness matrix K; the mass matrix M; the nodal forces vector F; the coordinates matrix COOR; the displacements dL; the connectivities matrix CN; the pressures at the surfaces P; the normals of the surfaces and the area of the surfaces.

The various data has been extracted using different methods that will be addressed in this chapter.

Extraction of dL

There is no need to modify the Kratos source code to obtain the displacements. In fact, the displacements are one of the most common results obtained from structural simulations. For that reason, the default GID problem type will already export them into a GID post-process format.

As explained, Kratos is a Python library written in C++ for speed, so the final user configures and executes the simulation via a high-level programming through Python, and this runs the fast C++ code. All the data is fed into Kratos through Python, and the data is then extracted through Python as well. If GID is used, a GID function that helps import and export the data can be employed.

In Python both the initial properties and the results are stored in

the *model_part* variable created by Kratos. Then, the results might be printed into a file using the function *gid_print.write_results* that takes as arguments: the *model_part*; an array with the results that need to be written, in this case, *["DISPLACEMENT"]*; the number of Gauss points; the current time; the current step; and an individual id for the file. Overall, the line should look like this:

Listing 1.1: Python example

```
gid_print.write_results(model_part,
general_variables.nodal_results,
general_variables.gauss_points_results,
current_time, current_step, current_id)
```

where *general_variables.nodal_results = ["DISPLACEMENT"]*.

Extraction of K, M and F

The K, M and F matrices are usually considered internal and thus, cannot be extracted using the method described in the previous section. That would be possible if the Kratos kernel exported the elemental matrices and vector to the Python level but this is not the case.

Simple extraction of K_{ll} and F_{ll}

There is, however, a simple way to extract the K and F matrix and vector. To do so, it is only necessary to change the Kratos echo level from 0 to 4. This echo level is meant for debugging and will print a lot of intermediate states and internal data to assist developers in finding bugs. Taking advantage of this function, it is possible to write the $K_l l$ matrix and the $F_l l$ in different files following the MatrixMarket file format.

This process, however, has two limitations. It does not extract the mass matrix and it is not possible to obtain the full K and F matrices - only the degrees of freedom are obtained. For that reason, a more powerful method is presented in the next section.

Extraction of K_e, M_e and F_e

To extract all matrices, it will be necessary to modify the Kratos C++ source. All the same, this process is made as simple as possible by the developers who understand that advanced usages of their product might need more than what is exposed to Python. Their solution has been to create a system of *applications* that can interact with the Kratos kernel at a lower level with C++.

This low-level access is required to extract the matrices, and so a new application has been created. This app will contain a single utility function that will be called from Python. This utility will take the *model_part* as input, like the gid_print function did in the previous section. In this case, however, it will be possible to access the full information contained in the *model_part* and not only the information exposed to Python because the utility is written in native C++. Among this newly accessible information, there are the elemental matrices of stiffness (K_e), the elemental matrices of mass (M_e) and the nodal forces vector (F_e).

To export the matrices, a simple loop that iterates over all the elements is used. At each iteration, a new line is written to a file using the *<fstream>* C++ library. Writing the file directly from C++ avoids the need to export the matrices to Python and the overall outcome is faster.

After the matrices have been extracted, a simple script is necessary to import them into Matlab. This Matlab script parses the file line by line (or

element by element) and uses a cell matrix to store the element number, the K_e, the M_e and the F_e on a single row per element; thus this matrix will have dimensions $4 \times number of elements$

Finally, the elemental matrices will need to be assembled into the system matrix of degrees of freedom, the K_{ll}, the M_{ll} and the F_{ll}. This process is described in the Section 1.2.

Extraction of COOR and CN

The coordinates matrix (COOR) and the connectivities matrix (CN) are input variables to Kratos and are directly exported by GID. It is possible to generate an ASCII file from GID with this information and parse it in Matlab to make them available.

Extraction of P, the normals, and areas

The pressure results from the fluid simulation are nodal variables. Since the nodes do not have surface nor normal vectors, the computation of the aerodynamic forces are not trivial.

For that reason, instead of working directly with the results of the fluid simulation, the boundary conditions applied to the structural problem are used. These boundary conditions are applied over surfaces, so they already have an area and normal vector. Moreover, it will be possible to generate new high-resolution structural simulations using the ROM results of the pressures if necessary. To do so, a new structural problem will be able to be formulated using the new pressures as boundary conditions.

The only caveat of using the boundary conditions instead of the nodal results is that some resolution is lost because there are fewer boundary

conditions than nodal results. Nevertheless, the results of the total aerodynamic forces should not vary much because the mean of the nodal pressures is used to calculate the surface pressure.

Since the CAD software applies the boundary conditions, the easiest way of obtaining the pressures, the area and the normal is to modify the exporter module in the CAD software. Moreover, the software already has functions that calculate the normal vector and the area of a surface, so the implementation is almost trivial.

The newly modified software generates a new file in the exported GID project that contains the pressures, the normals of each surface and it is area.

This text file can later be parsed in Matlab to obtain a matrix with the requested data.

1.2 Assemble Matrix

Basic algorithm

This section refers to the stiffness matrix K. Nevertheless, the same process is valid for the mass matrix M.

Assembling the matrix in Matlab presented a technical challenge because working with a full matrix would require too much memory. The size of the K matrix would be (nNodes * nDOF) by (nNodes * nDOF); the mesh in this project has 11073 nodes and 6 DOF per node, hence the array would have $(11073 \cdot 6)^2 = 4,414,007,844 positions$ if each number uses 64 bits, the total memory used by the array would be $4,414,007,844 * 64/8 = 35,312,062,752 bytes = 35.3 Gb$. However, the stiffness matrices contain a

vast amount of zeros and hence, it is inefficient to store every single element.

In order to store all the elements, the matrix is compressed using a system called "compressed row sparse matrix". This system uses three vectors to store only the elements different than 0. Two of the vectors (AJ and AI) are used to calculate the position of each value stored in the A vector. Using this system two vectors (A and AI) have a length equal to the number of elements different than 0; the other vector (AJ) has a length equal to the number of rows plus 1. This system is very efficient in terms of memory usage as well as computational requirements when doing common matrix operations. Matlab has its own implementation of this method built-in with the sparse matrix function.

There is, however, one drawback: the elements must be stored in order. And it is inefficient to retrieve the elements one by one. This, in turn, makes adding elements to an existing array very resource hungry. Since the elements must be stored in order, adding a new element requires retrieving the already stored elements to determine the right position in the vector for the new value. For this reason, adding new elements to an existing matrix is not recommended.

The solution, in this case, has been to store the values in three vectors (A, J and I). This is not, however, a compressed row sparse matrix since a position is created in every array for each element. Hence, this system uses more memory than a Matlab sparse matrix but does not require its values to be sorted and it is faster to operate with. Additionally, an addresses matrix has been built with the dimensions of nNodes by nNodes. This addresses matrix will become handy when an element must be added to a position that has already been registered in the vector. Instead of doing an expensive search through the vector to find the existing one, the addresses

matrix will be used to find the requested position.

The dimensions of the final stiffness matrix will be (nNodes * nDOF) by (nNodes * nDOF) and, as previously demonstrated, it must be stored as a sparse matrix. However, if we want to store the position where each element in the array is in our sparse vectors, it is not necessary to have a matrix that big. Considering that all the DOF of each node will be stored together, we can store only the position of the first DOF for each node and the rest DOF will be calculated from that. Thereby the address matrix will be nNodes by nNodes in size.

An example of an address matrix is illustrated in the Figure 1. In this figure two matrices are superposed, the real full K matrix and the address matrix. The small indices indicate the position of the full matrix and the big ones indicate the position of the addresses matrix. In this example, the problem has 2 nodes and 2 DOFs per node, hence the full stiffness matrix is 4 by 4 and the addresses, matrix is 2 by 2. The numbers inside the matrix indicate the indexes in our vectors where these values are stored. The addresses matrix will contain only the circled values and the rest will be calculated using the equation XX.

$$Ad(nodeI, nodeJ) + ((DOFi - 1) * 6 + DOFj) - 1 \tag{1.1}$$

In the equation 1.1 the $Ad(nodeI, nodeJ)$ function represent the address matrix and the indexes $(nodeI, nodeJ)$ are the nodes where the values must be stored. This part of the expression will return the position in the vectors of the first DOF for the requested combination of nodes (nodeI, nodeJ). The rest of the equation is needed to calculate the position of the requested DOF and the indices $(DOFi, DOFj)$ represent the combination

	j	**1**		**2**	
i		1	2	3	4
1	1	(9)	10	(5)	6
	2	11	12	7	8
2	3	(1)	2	(13)	14
	4	3	4	15	16

Figure 1: Example matrix, superposition of the full stiffness matrix (small indexes) and the address matrix (big indexes and circled values)

of DOFs requested inside that combination of nodes.

It must be noted that in a real case the vectors and the address matrix are filled while the element stiffness matrix is being red. As a result, the elements in the vectors will not be sorted. Also, not all the positions in the address matrix will be filled because not all the positions in the stiffness matrix will have values. Since the address matrix will contain lots of zeros too, it might be reasonable to use a sparse matrix. Nevertheless, if the memory is not a limitation a full matrix will produce much faster results. However, if a sparse matrix must be used, this system will still be faster than filling the final stiffness matrix directly because the address matrix is much smaller than the stiffness matrix and each position must only be edited the first time an element is added. If it is the second time that a combination of nodes is requested it will not be necessary to create a new address, however, it will still be necessary to add the new value to the existing one in the final stiffness matrix; and, as seen, this is very expensive to do directly in a sparse matrix.

It can be argued that this system uses more memory than needed because the position of each element in the values vector A is stored in the

vector J and I as well as in the address matrix. While it is true that more memory is used, this is done for a faster operation. The vectors J and I are necessary to create the K-sparse matrix faster, and the address matrix is necessary to build the vectors J, I and A. In the end, this system uses moderate memory and is very fast when assembling medium-large sparse matrices.

Modification to compute K_{ll} directly

In most cases the matrix that will be used in the end is not the stiffness matrix K but the stiffness matrix of the unrestricted nodes. This matrix is so-called K_{ll}. In order to speed up the generation of the K_{ll} matrix, the algorithm has been modified to generate the K_{ll} directly instead of the K matrix.

To do this some things must be taken into consideration:

Previously every time a new combination of nodes was requested, nDOF · nDOF positions were reserved in the values vectors. However, since now there are entire columns and rows missing it should be checked if the requested nodes have restricted DOFs and limit the amount of positions reserved in the vectors. This has been implemented with a series of conditionals.

Similarly, this must also be taken into account when calculating the positions i and j inside the final stiffness matrix. Previously it was possible to do $numberOfPreviousNodes \cdot nDOF$ but since now the number of DOFs per node are different in each node this is no longer true. The nDOFs must be computed per each previous node. In our case however, there are only two types of nodes: completely free (6 DOFs) and displacement locked (3 DOFs). Because of this its possible to implement it like this in Matlab:

Listing 1.2: Calculation of the previous number of coordinates

```
prevJCoords = (nodeJ - sum(r<=nodeJ))*6 + sum(r<=nodeJ)*3;
```

Also, if the current combination of nodes has restricted DOFs the computation of the position of each DOF inside the vectors will change. The equation 1.1 will no longer be valid. Now, it possible that the rows or columns are different than 6. In our case, there will be four possible equations depending on which node has restricted DOFs.

For i and j restricted:

$$Ad(nodeI, nodeJ) + ((DOFi - 4) * 3 + (DOFj - 3)) - 1 \tag{1.2a}$$

For i restricted:

$$Ad(nodeI, nodeJ) + ((DOFi - 4) * 6 + DOFj) - 1 \tag{1.2b}$$

For j restricted:

$$Ad(nodeI, nodeJ) + ((DOFi - 1) * 3 + (DOFj - 3)) - 1 \tag{1.2c}$$

For none restricted:

$$Ad(nodeI, nodeJ) + ((DOFi - 1) * 6 + DOFj) - 1 \tag{1.2d}$$

As seen, if there are none restricted nodes the equation is the same. The conditions used to trigger one or another equation are functions that return 1 or 0 depending on whether a node has or not restricted DOFs. Knowing this it would be possible to implement all this functionality in a single line of code. However, this line would be too large and too complicated to edit.

Finally, since the code loops for every element in each K_e matrix, there must be implemented a condition to determine whether a particular position of the Ke matrix must be included in the final K_{ll} matrix. In our case, it suffices to check wheter or not the nodeI or nodeJ are restricted and, if one is, check if its DOF is greater or smaller than 4 (remember that in this matrix a node is either free or has restricted the first 3 DOFs).

2 Test interpolation in the DOFs

Three different interpolation methods have been tested. A custom algorithm that worked really well for some specific cases -like the one shown on Figure 2- (the results from this method are marked in green circles in the figure); direct interpolation between the two nearest points (marked in red circles); and a regression using all the available data (marked in black circles). Additionally, in the figure the real values calculated using FE methods are indicated by a black cross.

2.1 The algorithm

The following text will use "HF points" or "HF data" to refer to the data obtained using the FE analysis and will use "requested point" or "requested value" to refer to the new angle of attack who's pressure value is not yet known.

Since the data seem to have a linear shape, the algorithm uses both a linear regression and a direct interpolation from the nearest two values. In order to decide how much of each method is used in the final interpolation, two variables are taken into account: the R^2 coefficient from the regression and the position of the requested value relative to the two nearest HF points. With this algorithm, the interpolation will coincide with the FE

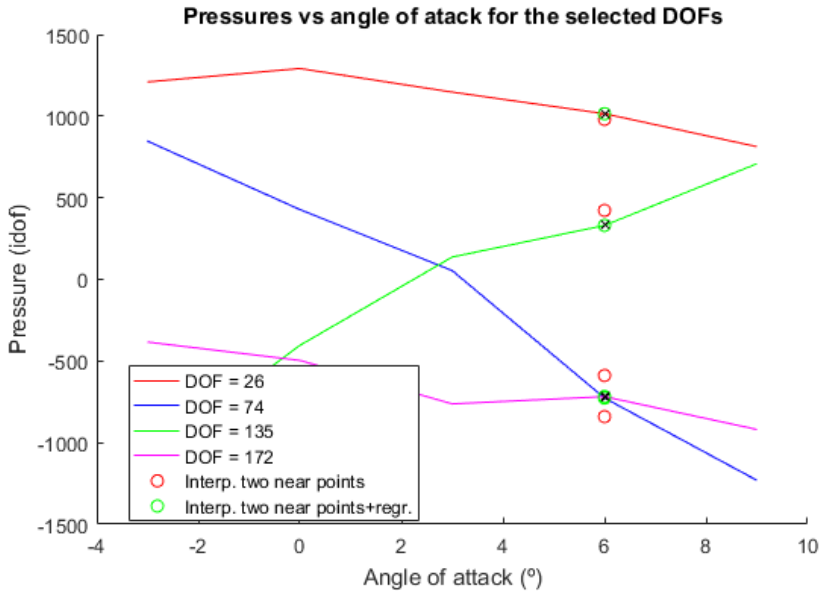

Figure 2: Plot of the selected DOF and their computed approximations for the 6º angle (without using the 6º value) using two different approches (red and green circles)

result if the required position is already calculated using the HF method; on the other hand, the algorithm will start to use more from the regression if the requested point is far away from any HF data. The idea behind this behavior is that the if a required location -for example at $x = 0.99$- is near a solution -for instance at $x = 1$-, the y-coordinate will probably be near too. All the same, when the requested point is farther away -for example at $x = 3$- the y-coordinate may have less to do with the nearest values -that in that case are not near- and more with the regression since maybe the calculated points are ”exception” from the general trend of the results.

The algorithm is described in the following equations from 2.1 to 2.4.

$$c = (x_i - x_1)/(x_2 - x_1) \tag{2.1}$$

where x_i is the desired abscissa coordinate, and x_1 and x_2 are the nearest points to the approximation; c ranges from 0 to 1 depending on the position of x_i inside the segment defined by x_1 and x_2.

$$k = abs(c - 0.5) * 2 \tag{2.2}$$

Since c is fenced between 0 and 1, k is delimited too by 0 and 1. The value of $k = 1$ if $c = 0.5$ (the farthest away from the points); and $k = 0$ if $c = 0$ or $c = 1$, ergo, if the desired value happens to be one of the points.

$$z = max(c, 1 - c) * k + (1 - R^a) * (1 - k) \tag{2.3}$$

z is the weight of the interpolated solution in the final equation 2.4. If $z = 1$, the final result will be exactly like an interpolation between the two nearest HF points; if $z = 0$, however, the result will be exactly like a regression using all the HF points available.

In the equation 2.3 R is the R^2 factor of the linear regression. The calculation of z is defined by the weighted mean between the max of c and a value that depends on R. The first factor -max of c- is defined between 1 and 0.5 and the second is defined between 0 and 1. The weighted average can then take values between 0 and 1. The parameter a can take different values, in this case is set to 10.

It's important to notice that if the $k = 0$ (that means the requested value is exactly in the middle of two nearest HF points), z will only depend on R (and, unless R = 1, the approximated value will be a mix of the interpolated result and the linear regression). If $k = 1 => z = 1$, thus the requested value is over one of the HF points and the result will be the based

Method	Error relative to the HF value
Interpolation	7.17%
Regression	7.82%
Algorithm	7.52%

Table 1: Results showing the mean errors using three interpolation methods

only on the interpolation. Therefore, the result will be exactly the value of the HF point.

$$finalAprox = z * interpolatedVal + (1 - z) * regresionVal; \qquad (2.4)$$

2.2 The tests

The algorithm seems to work very well on the previous image, however, after some testing, the best solution appeared to be the simple interpolation between the two nearest HF points. To state that, the three propositions have been tested against all known values, the mean error of each method compared to the HF value can be seen on table 1. The results show that the best method is the simple interpolation and the worse is the regression line with the algorithm falling in between.

3 Comprehensive list of available parameters and options in the CAD software

A comprehensive list of options has been listed below with its corresponding function and explanation.

- **Wing parameters**
  These parameters handle the exterior shape of the wing.

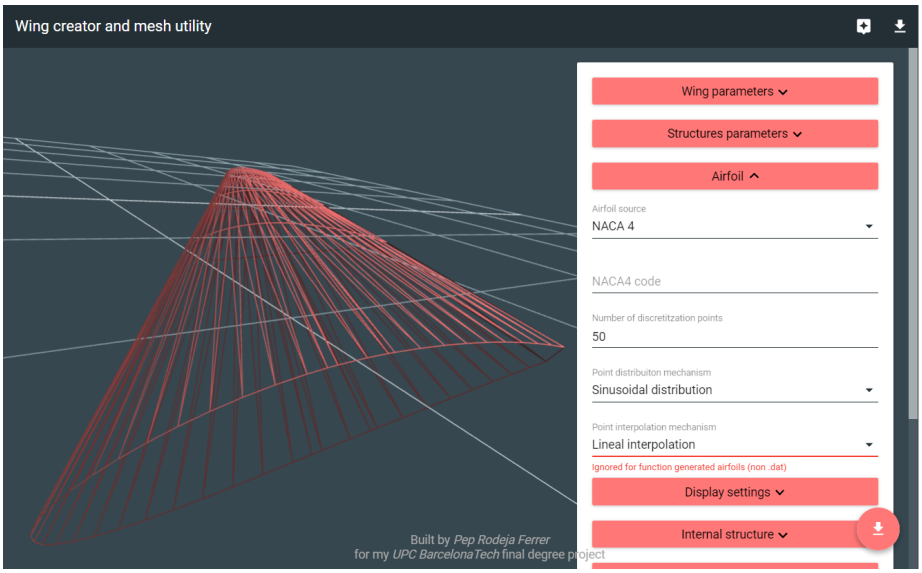

Figure 3: Screenshot of the software

  - Length
    The total span of the semiwing.
  - Root chord
    The chord of the airfoil at the root.
  - Tip chord
    The chord of the airfoil at the root.
  - Sweep angle
    Angle of the attack border with the line perpendicular to the plane fuselage.

- **Structure parameters**
  These parameters handle the internal structure of the wing.

  - Number of ribs
    The total number of ribs in the internal structure.
  - 1st Beam position
    The position of the first beam in percentage to the airfoil chord.
  - 2nd Beam position
    The position of the second beam in percentage to the airfoil chord.
  - Beam extension
    Length of the part of the beam that extends into the fuselage

- **Airfoil parameters**
  These parameters handle the airfoil and its discretitzation.

  - Airfoil source
    The source of the airfoil (NACA 4 equations or .dat files).
  - Airfoil name
    NACA4 name or name of the .dat file.

  - Number of discretization points
    The total number of discretization points per airfoil.
  - Point distribution mechanism
    Linear distribution or sinusoidal distribution for better resolution on the border of attack.

- **Display settings**
  These parameters handle how the wing is seen on the live 3D view. They have no effect on the exported geometry.

  - Internal structure
    * Visible
      Whether or not the internal structure is visible in the preview.
    * Visualization type
      Solid or wireframe. Wireframe example on Figure 3
  - External structure
    * Visible
      Whether or not the external structure is visible in the preview.
    * Visualization type
      Solid or wireframe. Wireframe example on Figure 3
  - Fluid box
    * Visible
      Whether or not the fluid box is visible on the preview.
    * Visualization type
      Solid or wireframe. Wireframe example on Figure 3

- **Internal structure**
  Change the element type fo the internal structure. Right now only shell can be selected.
  - Type
    Element type: Solid or shell. Only shell is suported now.
- **External structure**
  Change the element type fo the external structure. Right now only shell can be selected.
  - Type
    Element type: Solid or shell. Only shell is suported now.
- **Fluidbox parameters**
  Change the size and position of the fluid box.
  - Width
  - Length
  - Height
  - x coordinate
  - Angle of attack
- **Export settings**
  Change the groups that will be exported, the problem type and apply some boundary conditions.
  - GID problem type
    Problem types available are none, Kratos fluid and Kratos structural.

- Fluid simulation results
  Select the results of the fluid simulation to apply the resultant pressures as boundary conditions for the structural problem.
- GID mesh file
  Select an ASCII mesh file to use with the current configuration.
- Export separate vertices
  Whether or not the different objects (internal structure and external structure) should share the same vertices and segments if possible.
- Export external mesh
  Whether or not the external mesh should be exported.
- Export internal mesh
  Whether or not the internal mesh should be exported.
- Export fluid box
  Whether or not the fluid box should be exported.