Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Adrian Opheim

Enhancing the Analysis Possibilities of Fedem by Performing Model Reduction in ANSYS

Master's thesis in Produktutvikling og produksjon
Supervisor: Terje Rølvåg
June 2019

Adrian Opheim

Enhancing the Analysis Possibilities of Fedem by Performing Model Reduction in ANSYS

Master's thesis in Produktutvikling og produksjon
Supervisor: Terje Rølvåg
June 2019

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Abstract

When performing dynamic analyses, model reduction is a method used that will reduce the computational cost and storage requirements in finite element analysis programs. The dynamic analysis software Fedem uses the same model reduction techniques as the finite element analysis software ANSYS. As ANSYS has a wider range of element types available for analyses, Fedem would benefit from utilizing this when performing dynamic analyses.

To enable this integration between Fedem and ANSYS, the ANSYS ACT extension "ModRed" has been developed. It offers the possibility of calculating and exporting the matrices needed by Fedem for performing dynamic analyses. In order to test performance of the extension, CMS model reduction of the same models has been performed in both ANSYS and Fedem for investigating the equality of the resulting matrices.

The results show that the full mass matrices of both systems are nearly identical, but the reduced mass matrices show poor similarity, with 10% to 15% difference from the Fedem reference matrix. The calculated gravity vectors give very varied equality to the used Fedem reference; from being close to identical to having a 36.5% difference in the worse test case.

As an issue in partitioning ANSYS matrices has been discovered, and since this issue is believed to be a cause for the varied results, it is too early to make a conclusion on the accuracy acquired from the extension. More testing using a wider range of test models and element types, as well as resolving the issue of matrix partitioning is needed. Despite varying results in testing, the ModRed extension has proven to be a valuable and easy-to-use method for enhancing the analysis capabilities of Fedem.

Sammendrag

Ved utførelse av dynamisk analyse er modellreduksjon en metode ofte tatt i bruk. Denne metoden vil redusere utregningstiden og lagringsbehovene for elementmetode-program. Dynamisk analyse-programvaren Fedem benytter de samme teknikkene for modellreduksjon som det som blir brukt i elementmetode-programmet ANSYS. Ettersom ANSYS har et større utvalg av elementtyper tilgjengelig, vil det kunne være nyttig for Fedem å utnytte dette i dynamiske analyser.

ANSYS ACT-programtillegget ”ModRed” er blitt utviklet for å gjøre en slik integrering mellom ANSYS og Fedem mulig. Programtillegget gjør det mulig å regne ut og eksportere de nødvendige matrisene som skal til for at Fedem skal kunne utføre dynamiske analyser. For å teste ytelsen til programtillegget, har en CMS-modellreduksjon av identiske modeller blitt utført i både ANSYS og Fedem, og de resulterende matrisene er blitt sammenliknet.

Resultatene viser at de fulle massematrisene fra begge systemer er så godt som identiske. Det viser seg imidlertid at de reduserte massematrisene kun i liten grad er like, med 10% til 15% forskjell fra Fedems referansematrise. De utregnede gravitasjonsvektorene viser svært varierende likhet til referansen fra Fedem; fra å være tilnærmet identiske, til å være 36.5% forskjellige i tilfellet med størst forskjell.

Ettersom et problem med partisjonering av ANSYS-matriser har blitt oppdaget, og siden dette antas å være en årsak til de varierende resultatene, er det for tidlig å fastslå nøyaktigheten man kan oppnå ved å bruke programtillegget. Ytterligere testing med bruk av et større utvalg test-modeller og element-typer, samt å rette opp i matrise-partisjonerings-problemet er nødvendig. Til tross for varierende resultater ved testing, har ModRed-programtillegget vist seg å være en verdifull metode som det er lett å ta i bruk for å utvide Fedems muligheter for analyse.

Preface

This master's thesis is the product of work done during the spring term in 2019 as the final part of a five-year master's degree in mechanical engineering. The thesis is written at the Norwegian University of Science and Technology (NTNU) in Trondheim, at the Department of Mechanical and Industrial Engineering.

The project was chosen based on my own interest in both mechanical engineering and software development. As this project would allow me to explore both fields, the potential learning outcome was a key driver. I would like to thank my supervisor, Professor Terje Rølvåg from the Department of Mechanical and Industrial Engineering for introducing me to the project and providing constructive feedback. In addition, I would like to thank Knut Morten Okstad from Fedem Technology for valuable support on the Fedem software.

Trondheim, June 2019

Adrian Opheim

Table of Contents

Abstract **i**

Sammendrag **ii**

Preface **iii**

Table of Contents **vi**

List of Tables **vii**

List of Figures **ix**

Abbreviations **x**

1 Introduction **1**

2 Theory **3**
2.1 The Finite Element Method in Linear Dynamics 3
2.1.1 Assembling elements to the global structure 4
2.2 Reduction Methods for Dynamic Problems 6
2.2.1 Guyan Reduction 8
2.2.2 CMS Model Reduction 9
2.3 Using ANSYS for Solving Finite Element Problems 10
2.3.1 ANSYS Parametric Design Language (APDL) 10
2.3.2 Performing Model Reduction in ANSYS 12
2.3.3 Accuracy of Model Reduction Performed in ANSYS 15
2.3.4 ANSYS ACT 17
2.3.5 Structure of an ACT Application 17
2.3.6 ANSYS Element Library 18
2.3.7 Degree of Freedom Ordering 21
2.4 Fedem 22
2.4.1 Model Reduction in Fedem 23
2.4.2 Importing reduced models to Fedem 23
2.4.3 Gravity Vectors in Fedem 24

2.4.4 The FFQ4 Shell Element in Fedem . . . 26
2.4.5 Degree of Freedom Ordering in Fedem . . . 27
2.5 Software Development . . . 28
2.5.1 Validating Software by Unit Testing . . . 29

3 Method **31**
3.1 The ModRed Extension for Performing CMS Model Reduction in ANSYS 32
3.1.1 Installation . . . 32
3.1.2 Importing a model into ANSYS Mechanical . . . 33
3.1.3 Defining Settings for the ModRed Extension . . . 35
3.1.4 The APDL Commands Written by ModRed . . . 36
3.1.5 Matrix Export . . . 37
3.1.6 Generation of files to be read by Fedem . . . 38
3.2 Verifying Data From the ModRed Extension . . . 38

4 Results **43**
4.1 Similarity of Mass Matrices between ANSYS and Fedem . . . 44
4.2 Testing for Consistent Mass in Models . . . 46
4.3 Calculation of Gravity Vectors . . . 46
4.4 Replicating Fedem Gravity Vectors . . . 48
4.5 Generation of FTL Files in ANSYS . . . 48
4.6 Screencast Demonstration of the ModRed Extension . . . 49

5 Discussion and Future Work **51**
5.1 Instability of Results . . . 51
5.2 Future Work . . . 54

6 Conclusion **57**

References **59**

Appendix **63**

A Source Code for the ModRed ACT Extension **63**
A.1 ModRed.py . . . 63
A.2 ModRed.xml . . . 87
A.3 test_ModRed.py . . . 89

List of Tables

2.1 Accuracy of CMS reduced model in ANSYS compared to the full model . 16
2.2 Elapsed time for applying CMS model reduction and solving the model in ANSYS 16
2.3 Syntax of Technology Link Format 24

3.1 Supported mesh file formats for import to ANSYS Mechanical 34

4.1 FE models used for testing the ModRed extension 43
4.2 Results from testing for inconsistent mass 46
4.3 Results from replicating Fedem gravity vectors 48

List of Figures

2.1 Bar in axial deformation . . . 3
2.2 Assembly of the global matrices . . . 6
2.3 Modal analysis in ANSYS Workbench . . . 11
2.4 Translation of actions performed in ANSYS Mechanical to APDL code . . . 12
2.5 Structure simplified into finite elements . . . 13
2.6 Automotive suspension system used for testing CMS reduction accuracy . . . 16
2.7 Formats for sharing ACT extensions . . . 18
2.8 The four main element types . . . 19
2.9 The SHELL181 elemet . . . 20
2.10 FFQ4 element in Fedem . . . 26
2.11 Incremental development . . . 29

3.1 The ModRed extension loaded into the toolbar . . . 32
3.2 The ModRed extension in ANSYS Mechanical . . . 33
3.3 Importing an existing mesh into ANSYS Mechanical . . . 33
3.4 Import a Native Model . . . 34
3.5 The ModRed extension in ANSYS Mechanical . . . 35
3.6 Node selection tool . . . 35

4.1 The small test model twoQUAD4 . . . 44
4.2 Percentage difference of mass matrices from Fedem and ANSYS . . . 45
4.3 Relative difference of gravity vectors calculated from the reduced mass matrix . . . 47
4.4 Visualization of twoQUAD4 element in Fedem . . . 50

Abbreviations

ACT	=	ANSYS Customization Toolkit
APDL	=	ANSYS Parametric Design Language
CAD	=	Computer Aided Design
CAE	=	Coumpter Aided Engineering
CMS	=	Component Mode Synthesis
DOF	=	Degree of Freedom
FE	=	Finite Element
FEM	=	Finite Element Method
GUI	=	Graphical User Interface
WBEX	=	WorkBench Extension
XML	=	Extensible Markup Language

Chapter 1

Introduction

When performing dynamic analyses of structures, similar repeated matrix calculations must be performed for every time step in the analysis. Dynamic analyses are therefore generally much more expensive in terms of computational time and storage space than performing static analyses. In order to reduce the cost of performing dynamic analyses, model reduction methods that reduce the number of degrees of freedom in the system may be applied. Guyan reduction and the component mode synthesis method are examples of such methods. The finite element software ANSYS have both these methods implemented in its APDL solver. In order to access these methods through the ANSYS Mechanical software, an ANSYS ACT extension must be developed, as model reduction techniques are currently not available through ANSYS Mechanical.

The dynamic analysis software Fedem uses the same model reduction techniques as ANSYS. Due to the extensive finite element library available, it is of interest to perform model reduction in ANSYS. The reduced models may thereafter be imported to Fedem for further dynamic analyses. This thesis will present such an extension that enables this integration.

Chapter 2

Theory

2.1 The Finite Element Method in Linear Dynamics

When analyzing continuous problems, very few of them have an exact, closed-form solution, implicating that discretization techniques should be applied in order to make an approximated system. The behavior of the discretized system could then be computed and used as an approximation for the continuous system. The finite element method is an discretization method that has become the preferred computational method for analyzing structures. Dividing the a structure into a finite number of elements, then assembling them while keeping continuity in the displacements of connected elements for representing the global displacements is the core principle of the finite element method.

For illustrating the principle of the finite element method, we consider a bar of length L in axial deformation subjected to axial loads P_1 and P_2. The bar is divided into N finite elements of length l, as shown in Figure 2.1.

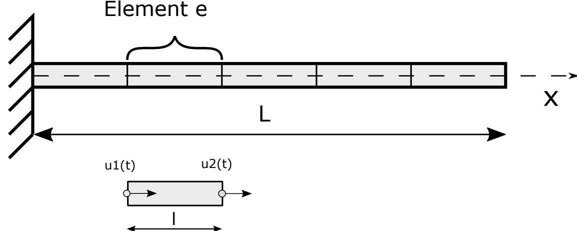

Figure 2.1: Bar in axial deformation

For each element, the displacement field $u(x, t)$ is linearly interpolated by

$$u(x, t) = u_1(t)\phi_1(x) + u_2(t)\phi_2(x) \tag{2.1}$$

where $\phi_1(x)$, $\phi_2(x)$ are so-called shape functions of the element. The shape functions are chosen in such a way that they fullfill the boundary conditions

$$u(0,t) = u_1(t) \qquad u(l,t) = u_2(t)$$

u_1 and u_2 are axial displacements at the element ends, referred to as nodes.

For the bar in axial deformation, the shape functions are linear, given as

$$\phi_1(x) = 1 - \frac{1}{l} \qquad \phi_2(x) = \frac{x}{l} \tag{2.2}$$

(2.1) can be written in matrix form

$$u(x,t) = \mathbf{F}_e(x)\mathbf{q}_e(t) \tag{2.3}$$

where x must be within the element e, and

$$\begin{aligned} \mathbf{F}_e(x) &= \begin{bmatrix} \phi_1(x) & \phi_2(x) \end{bmatrix} \\ \mathbf{q}_e(t) &= \begin{bmatrix} u_1(t) & u_2(t) \end{bmatrix}^T \end{aligned} \tag{2.4}$$

Using this, we can compute the elementary mass and stiffness matrices $\mathbf{M}_e$ and $\mathbf{K}_e$ given by

$$\begin{aligned} \mathbf{M}_e &= \int_0^l m\mathbf{F}_e^T\mathbf{F}_e dx \\ \mathbf{K}_e &= \int_0^l EA\frac{d\mathbf{F}_e^T}{dx}\frac{d\mathbf{F}_e}{dx}dx \end{aligned} \tag{2.5}$$

For the axial bar, we obtain the matrices

$$\mathbf{K}_e = \frac{EA}{l}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \quad \mathbf{M}_e = \frac{ml}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \tag{2.6}$$

The force element vector $\mathbf{g}_e$ is equal to to the end loads of the element:

$$\mathbf{g}_e^{(1)} = \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} \tag{2.7}$$

2.1.1 Assembling elements to the global structure

For expressing dynamic equilibrium for whole bar, we need to know the topology of every element e: Where is it located with respect to the global model and with respect to the other elements of the structure? This is solved by a topology matrix $\mathbf{L}_e$. It is constructed so that

$$\mathbf{q}^T = \mathbf{L}_e\mathbf{q} \tag{2.8}$$

where $\mathbf{q}$ is a matrix containing all $(N+1)$ nodal displacements:

$$\mathbf{q} = \begin{bmatrix} u_0 & u_1 & u_2 \cdots u_N \end{bmatrix}^T$$

For instance, element 1 and 2 of Figure 2.1 have the topology matrix

$$\begin{aligned} \mathbf{L}_1 &= \begin{bmatrix} 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \end{bmatrix} \\ \mathbf{L}_2 &= \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \end{bmatrix} \end{aligned} \tag{2.9}$$

Using the topology matrix, we can assemble all N elements into a global system, and we can then define

- The mass matrix of the assembled system:

$$\mathbf{M} = \sum_{e=1}^{N} \mathbf{L}_e^T \mathbf{M}_e \mathbf{L}_e \tag{2.10}$$

- The stiffness matrix of the assembled system:

$$\mathbf{K} = \sum_{e=1}^{N} \mathbf{L}_e^T \mathbf{K}_e \mathbf{L}_e \tag{2.11}$$

- The load vector of the assembled system:

$$\mathbf{g} = \sum_{e=1}^{N} \mathbf{L}_e^T \mathbf{g}_e \tag{2.12}$$

This corresponds to an assembly as seen in Figure 2.2. From the figure, one can observe that

- The diagonal terms of the stiffness and mass matrix add up two and two along the diagonal of the matrix.

- The shaded zone correspond to the clamped end of the bar, and could therefore be set to 0.

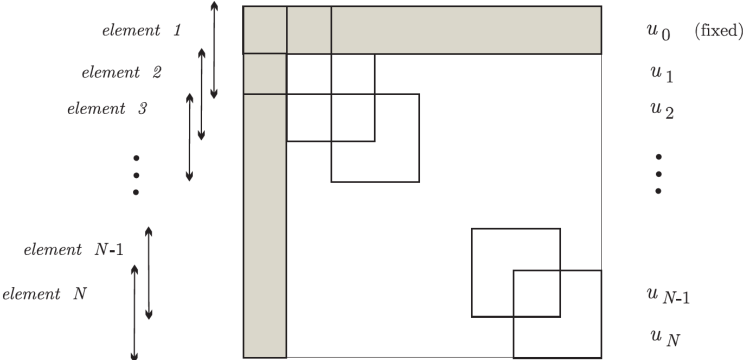

Figure 2.2: Assembly of the global matrices

For the axially loaded bar, the global structural matrices become

$$\mathbf{K} = \frac{EA}{l} \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & \mathbf{0} & \\ & -1 & 2 & \ddots & & \\ & & \ddots & \ddots & -1 & \\ & \mathbf{0} & & -1 & 2 & -1 \\ & & & & -1 & 1 \end{bmatrix} \tag{2.13}$$

$$\mathbf{M} = \frac{ml}{6} \begin{bmatrix} 4 & 1 & & & & \\ 1 & 4 & 1 & & \mathbf{0} & \\ & 1 & 4 & \ddots & & \\ & & \ddots & \ddots & 1 & \\ & \mathbf{0} & & 1 & 4 & 1 \\ & & & & 1 & 2 \end{bmatrix} \tag{2.14}$$

2.2 Reduction Methods for Dynamic Problems

When a finite element analysis of a static system is performed, static deformations and stress levels in small details of the system is usually of interest. The static FEM models will therefore usually have a highly refined mesh and a corresponding very high number of degrees of freedom. One are able to solve these models fairly efficiently with static solvers, but if one wishes to find the dynamic behavior of the system, the computing time required is often unacceptable. However, the highly refined meshes used in static analyses are often not needed to capture the dynamic behavior of the system. If one were to find the free vibration modes of the system, we know that the first free vibration modes have a rather smooth deformation, meaning that a coarser mesh would be sufficient to capture these

modes. The reduction methods for dynamic problems addresses this problem. Because creating a fine mesh that captures the required static solutions is an important and time-consuming part of FE analysis, the reduction methods do not modify the fine mesh created for static analyses, but instead reduces the size of the dynamic problem to solve.

Model reduction can in general be expressed as

$$\mathbf{v} = \mathbf{H}\mathbf{q} \tag{2.15}$$

where $\mathbf{v}$ is the full set of degrees of freedom for the fine mesh and is of size $(n \times 1)$. $\mathbf{H}$ is the reduction matrix and $\mathbf{q}$ is the reduced set of displacements used for capturing the dynamic behavior and is of size $(m \times 1)$. The aim for the model reduction is to achieve $m << n$ without a significant loss in accuracy for the stress results.

The matrix equation that governs system dynamics is expressed as

$$\mathbf{M}\ddot{\mathbf{v}} + \mathbf{K}\mathbf{v} = \mathbf{Q} \tag{2.16}$$

if one neglects damping. When reduction methods are applied to the system equations, the equation is partitioned as follows

$$\begin{bmatrix} \mathbf{M}_{ee} & \mathbf{M}_{ei} \\ \mathbf{M}_{ie} & \mathbf{M}_{ii} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{v}}_e \\ \ddot{\mathbf{v}}_i \end{bmatrix} + \begin{bmatrix} \mathbf{K}_{ee} & \mathbf{K}_{ei} \\ \mathbf{K}_{ie} & \mathbf{K}_{ii} \end{bmatrix} \begin{bmatrix} \mathbf{v}_e \\ \mathbf{v}_i \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_e \\ \mathbf{Q}_i \end{bmatrix} \tag{2.17}$$

where subscript e represents external nodes and subscript i represents internal nodes. External nodes in the model will typically be chosen to be joints, springs, dampers, external loads, control input, points of interest, etc. There should be as few as possible external nodes to reduce the simulation model as much as possible. As will be shown, the reduction methods will eliminate the internal nodes from the FE model.

From (2.17), the stiffness relation for a sub-structure can be written as

$$\begin{bmatrix} \mathbf{K}_{ee} & \mathbf{K}_{ei} \\ \mathbf{K}_{ie} & \mathbf{K}_{ii} \end{bmatrix} \begin{bmatrix} \mathbf{v}_e \\ \mathbf{v}_i \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_e \\ \mathbf{Q}_i \end{bmatrix} \tag{2.18}$$

Writing out the equations gives

$$\begin{aligned} \mathbf{K}_{ee}\mathbf{v}_e + \mathbf{K}_{ei}\mathbf{v}_i &= \mathbf{Q}_e \\ \mathbf{K}_{ie}\mathbf{v}_e + \mathbf{K}_{ii}\mathbf{v}_i &= \mathbf{Q}_i \end{aligned} \tag{2.19}$$

Solving the second equation for $\mathbf{v}_i$ gives

$$\begin{aligned} \mathbf{v}_i &= \mathbf{K}_{ii}^{-1}\mathbf{Q}_i - \mathbf{K}_{ii}^{-1}\mathbf{K}_{ie}\mathbf{v}_e \\ &= \mathbf{K}_{ii}^{-1}\mathbf{Q}_i + \mathbf{B}\mathbf{v}_e \end{aligned}$$

where $\mathbf{B}$ is denoted as the influence matrix, given by $\mathbf{B} = -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ie}$.

The internal displacements $\mathbf{v}_i$ can then be expressed as

$$\mathbf{v}_i = \mathbf{v}_i^{\text{dyn}} + \mathbf{v}_i^{\text{stat}} \tag{2.20}$$

where $\mathbf{v}_i^{\text{dyn}}$ represents internal displacements with external DOFs fixed, the dynamic part of the internal node's displacements

$$\mathbf{v}_i^{\text{dyn}} = \mathbf{K}_{ii}^{-1}\mathbf{Q}_i \tag{2.21}$$

and $\mathbf{v}_i^{\text{stat}}$ represents internal displacements as a function of external displacements, called the "static" part, because the internal nodes respond quasi-statically to the external nodes' displacements.

$$\mathbf{v}_i^{\text{stat}} = -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ie}\mathbf{v}_e = \mathbf{B}\mathbf{v}_e \tag{2.22}$$

When applying model reduction techniques, two methods are commonly used, namely Guyan reduction and Component Mode Synthesis (CMS). The two methods differ in how they are expressing the internal displacements $\mathbf{v}_i$. When applying Guyan reduction, the "dynamic" part $\mathbf{v}_i^i$ is neglected, and the reduction is based purely on the static response of the internal nodes from displacements at the external nodes, while the Component Mode Synthesis method uses both the dynamic and static part of the internal nodes' displacements

2.2.1 Guyan Reduction

Guyan reduction utilizes the fact that the quasi-static response of the internal nodes is often sufficient for describing the substructure's displacements. The reduction method neglects the dynamic part of the internal node's displacements, thus setting $\mathbf{v}_i^{\text{dyn}} = \mathbf{0}$. It is also assumed that no forces are applied on the internal nodes, $\mathbf{Q}_i = \mathbf{0}$. The reduction is then built by

$$\mathbf{v} = \begin{bmatrix} \mathbf{v}_e \\ \mathbf{v}_i \end{bmatrix} = \mathbf{H}_{\text{Guyan}}\mathbf{v}_e = \begin{bmatrix} \mathbf{I} \\ \mathbf{B} \end{bmatrix} \mathbf{v}_e \tag{2.23}$$

where $\mathbf{I}$ is the identity matrix, and $\mathbf{B} = -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ie}$ is the influence matrix.

The reduced stiffness and mass matrices from the Guyan reduction, respectively $\mathbf{k}_{\text{Guyan}}$ and $\mathbf{m}_{\text{Guyan}}$, are then found by premultiplying (2.16) with $\mathbf{H}_{\text{Guyan}}^T$ and inserting (2.23) for $\mathbf{v}$. The reduced stiffness and mass matrices are then

$$\begin{aligned} \mathbf{k}_{\text{Guyan}} &= \mathbf{H}_{\text{Guyan}}^T\mathbf{K}\mathbf{H}_{\text{Guyan}} \\ &= \mathbf{K}_{ee} - \mathbf{K}_{ei}\mathbf{K}_{ii}^{-1}\mathbf{K}_{ie} \\ \mathbf{m}_{\text{Guyan}} &= \mathbf{H}_{\text{Guyan}}^T\mathbf{M}\mathbf{H}_{\text{Guyan}} \\ &= \mathbf{M}_{ee} - \mathbf{M}_{ei}\mathbf{K}_{ii}^{-1}\mathbf{K}_{ie} - \mathbf{K}_{ei}\mathbf{K}_{ii}^{-1}\mathbf{M}_{ie} + \mathbf{K}_{ei}\mathbf{K}_{ii}^{-1}\mathbf{M}_{ii}\mathbf{K}_{ii}^{-1}\mathbf{K}_{ie} \end{aligned} \tag{2.24}$$

Using this, the dynamic equation for the whole system (2.16) can be reduced to the following for an undamped system

$$\mathbf{m}_{\text{Guyan}}\ddot{\mathbf{v}}_e + \mathbf{k}_{\text{Guyan}}\mathbf{v}_e = \mathbf{Q}_e \tag{2.25}$$

This results tells us that the whole system now is being represented only by the external node's accelerations, displacements and loads. Applying the Guyan reduction technique will therefore reduce the number of degrees of freedom in the system drastically.

If the Guyan reduction is applied to static problems, the exact solution is found. It is also computationally efficient, thus being a widely used reduction technique. However, if the structure has a dynamic behaviour, an approximation is introduced by neglecting the dynamic response of the internal nodes.

2.2.2 CMS Model Reduction

The Component Mode Synthesis (CMS) model reduction was introduced by Craig and Bampton [1] in 1968. The method is therefore also called the "Craig-Bampton Method". When applying CMS reduction, the term $\mathbf{v}_i^{\text{dyn}}$ is no longer neglected. Instead it is approximated by a linear combination of eigenvalues. In the following, we assume that we originally have n degrees of freedom. By partitioning through (2.17), we get p external and $n-p$ internal degrees of freedom.

In order to approximate $\mathbf{v}_i^{\text{dyn}}$, we remember that $\mathbf{v}_i^{\text{dyn}}$ correspond to the displacements in the substructure when the external degrees of freedom are fixed. This corresponds the the case where

$$\mathbf{Q}_i = \mathbf{v}_e = \mathbf{0} \tag{2.26}$$

Inserting this into (2.17) gives the equation

$$\mathbf{M}_{ii}\ddot{\mathbf{v}}_i^{\text{dyn}} + \mathbf{K}_{ii}\mathbf{v}_i^{\text{dyn}} = \mathbf{0} \tag{2.27}$$

When considering simple harmonic motion, the displacement $\mathbf{v}_i^{\text{dyn}}$ may be expressed as

$$\mathbf{v}_i^{\text{dyn}} = \boldsymbol{\phi}\sin\omega t \tag{2.28}$$

where $\boldsymbol{\phi}$ is the eigenvector defined by the eigenvalue problem

$$(\mathbf{K}_{ii} - \omega^2\mathbf{M}_{ii})\boldsymbol{\phi} = \mathbf{0} \tag{2.29}$$

It is possible to describe an arbitrary displacement as a linear combination of the $(n-p)$ eigenmodes. Using a selection s of the eigenmodes, $\mathbf{v}_i^{\text{dyn}}$ can be approximated as

$$\mathbf{v}_i^{\text{dyn}} = \sum_{k=1}^{s}\boldsymbol{\phi}_k y_k = \boldsymbol{\Phi}\mathbf{y} \qquad s < n-p \tag{2.30}$$

where

$$\boldsymbol{\Phi} = \begin{bmatrix}\boldsymbol{\phi}_1 & \boldsymbol{\phi}_2 & \cdots & \boldsymbol{\phi}_s\end{bmatrix} \tag{2.31}$$

is the eigenvector matrix with dimensions $(n-p)\times s$.

Using this result, the displacement vector $\mathbf{v}$ can be expressed as

$$\mathbf{v} = \begin{bmatrix}\mathbf{v}_e\\ \mathbf{v}_i\end{bmatrix} = \begin{bmatrix}\mathbf{I} & \mathbf{0}\\ \mathbf{B} & \boldsymbol{\Phi}\end{bmatrix}\begin{bmatrix}\mathbf{v}_e\\ \mathbf{y}\end{bmatrix} = \mathbf{H}_{\text{CMS}}\begin{bmatrix}\mathbf{v}_e\\ \mathbf{y}\end{bmatrix} \tag{2.32}$$

When combining the substructure dynamic equation (2.17) with the time derivatives of (2.32) and pre-multiplying with $\mathbf{H}_{\text{CMS}}^T$ we get

$$\begin{bmatrix} \mathbf{m}_{11} & \mathbf{m}_{12} \\ \mathbf{m}_{21} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{v}}_e \\ \ddot{\mathbf{y}} \end{bmatrix} + \begin{bmatrix} \mathbf{k}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{k}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{v}_e \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1 \\ \mathbf{q}_2 \end{bmatrix} \tag{2.33}$$

where

$$\begin{aligned} \mathbf{m}_{11} &= \mathbf{M}_{ee} + \mathbf{B}^T\mathbf{M}_{ie} + \mathbf{M}_{ei}\mathbf{B} + \mathbf{B}^T\mathbf{M}_{ii}\mathbf{B} \\ \mathbf{m}_{12} &= \mathbf{m}_{21}^T = \mathbf{M}_{ei}\mathbf{\Phi} + \mathbf{B}^T\mathbf{M}_{ii}\mathbf{\Phi} \\ \mathbf{k}_{11} &= \mathbf{K}_{ee} + \mathbf{K}_{ie}^T\mathbf{B} \\ \mathbf{k}_{22} &= \begin{bmatrix} \omega_1^2 & & & \\ & \omega_2^2 & & \\ & & \ddots & \\ & & & \omega_{n-p}^2 \end{bmatrix} \\ \mathbf{q}_1 &= \mathbf{Q}_e + \mathbf{B}^T\mathbf{Q}_i \\ \mathbf{q}_2 &= \mathbf{\Phi}^T\mathbf{Q}_i \end{aligned} \tag{2.34}$$

For the $\mathbf{k}_{22}$ matrix, $\omega_1^2, \omega_2^2, \ldots, \omega_{n-p}^2$ are the eigenvalues corresponding to the eigenmodes of eigenvector matrix $\mathbf{\Phi}$.

The substructure will be fully represented - meaning that no approximations are introduced - if all eigenmodes of the substructure is taken into account, thus setting $s = n-p$. Setting $s = 0$ will give the Guyan reduction presented in Section 2.2.1. When applying CMS reduction, one tries to keep s as small as possible for reducing the problem size. A criteria often used in practice is to include all eigenmodes having frequencies up to 1.8 or 2 times the highest frequency one wants to compute in the global structure. [2]

2.3 Using ANSYS for Solving Finite Element Problems

ANSYS is an American company founded in 1970 which develops and sells finite element simulation software. Their key product, the Mechanical APDL program, has the analysis capability to solve static and dynamic structural analyses, steady-state and transient heat transfer problems, mode-frequency and buckling eigenvalue problems, static or time-variying magnetic analyses, and various types of field and coupled-field applications.

2.3.1 ANSYS Parametric Design Language (APDL)

The Mechanical APDL program uses a command-line syntax called ANSYS Parametric Design Language (APDL) for writing commands to the FE solver. The language offers the possibility to use features like repeating commands, macros, if-then-else-branching, for-loops and matrix operations with APDL Math. In the last years, Mechanical APDL

has been replaced by, among others, the new products ANSYS Workbench and ANSYS Mechanical. However, all new ANSYS products still use the the FE solver implemented in Mechanical APDL. Learning to use the APDL language for writing FE solver commands is therefore of great importance. An example of APDL syntax can be seen in Section 2.3.2 and in the `onExportData()` method of `ModRed.py`, Section A.1. However, specifics of the APDL syntax will not be presented here. The Mechanical APDL Element Reference [3] serves as a great guide for referencing specific commands.

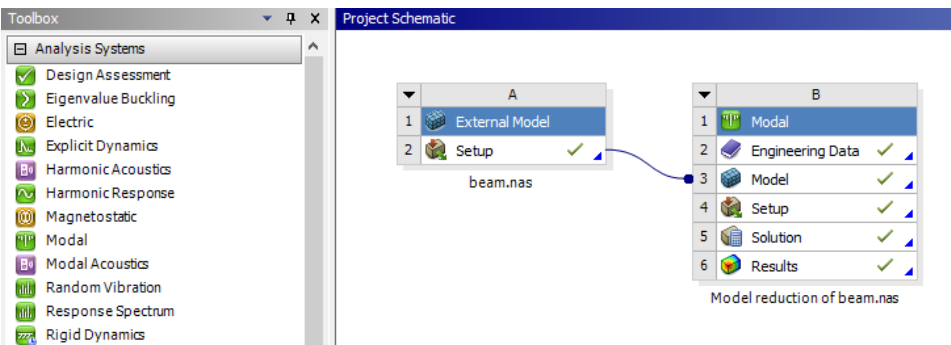

Figure 2.3: Example of a modal analysis set up in ANSYS Workbench that will be performed in the ANSYS Mechanical program.

Figure 2.3 shows the ANSYS Workbench environment that is used to analyse a simple beam constructed by shell elements. The ANSYS Workbench environment makes it easy to perform multi-domain analyses by dragging-and-dropping the analysis type seen to the left. When an analysis is opened, the user is taken to the ANSYS Mechanical program, where the analyses will be performed. When an analysis is correctly defined and the user hits ”Solve”, the following happens:

1. All items in the ANSYS Mechanical tree are converted to APDL commands, and written to the file `ds.dat` located in the ANSYS solver directory, illustrated in Figure 2.4.

2. Mechanical APDL is invoked in the background, performs the commands defined in the `ds.dat` file and returns results written in the file `file.rst`

3. The result file is read by ANSYS Mechanical, and displayed graphically to the user.

This means that it would be possible to write custom APDL commands directly to the `ds.dat` file for solving your specific problem. This is what is utilised in the ModRed application, Section 3.1

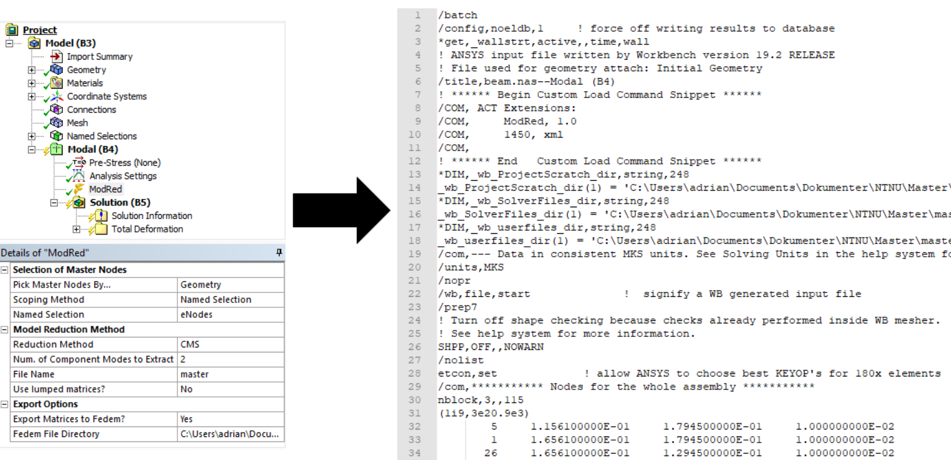

Figure 2.4: When the user hits "Solve" in ANSYS Mechanical, all analysis settings are translated to APDL code and sent to APDL in the background.

2.3.2 Performing Model Reduction in ANSYS

Model reduction is currently (Version 19.2) not available in the ANSYS Mechanical program. However, multiple model reduction techniques are available through APDL commands, among them Guyan (static) reduction and CMS reduction. Guyan reduction could be applied to both structural and non-structural analysis types. If Guyan reduction is applied to non-structural analyses, the $\mathbf{K}$ and $\mathbf{M}$ matrices are no longer expressing stiffness and mass quantities. Instead, they are matrices representing zero order terms ($\mathbf{K}$) and second order terms ($\mathbf{M}$). The CMS reduction implemented in APDL is not as versatile, and can only be applied to structural analyses. When performing model reduction in ANSYS, the process is divided into three separate steps, namely the generation pass, use pass and expansion pass.

The Generation Pass

The generation pass could further be divided into two parts: model generation and superelement generation.

The **model generation** part involves defining the element types, material properties, model geometry and specifying the jobname for reuse in the later use and expansion passes. If the model is already defined, as will be the case in the ModRed application, this part would involve importing the model and its properties correctly into the ANSYS environment.

The **superelement creation** part involves condensing the full model into one or multiple superelements. The superelement is defined by selecting external (master) nodes that will serve as the interface between other elements and the superelement. The master nodes

need to be defined even though the superelement is not to be connected to other non-superelements. Master nodes need to be selected with care, as forces or constraints will later be applied only to the master nodes. The master nodes are therefore typically selected to be at the joints or extremities of the full model. Figure 2.5 shows how one can model a bridge using super elements. It is modeled with bar elements taking axial deformation, as described in Section 2.1. By selecting green nodes as external nodes, 9 super elements can be created, one for every truss in the bridge. Their ID is marked with a square in Figure 2.5. The external nodes are selected at the joints, as this is where forces or constraints will be applied later. The external nodes' ID is marked with a circle.

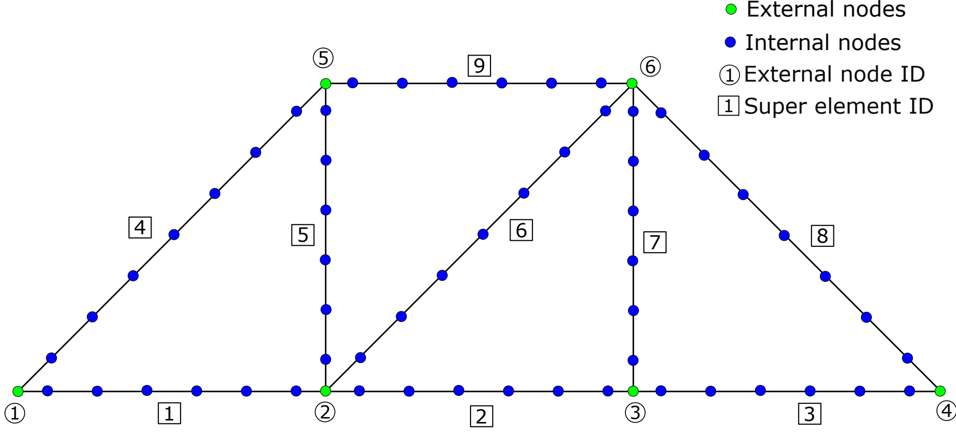

Figure 2.5: Structure simplified into finite elements

When the `solve` command is issued at the generation pass, multiple files are generated. Some are written in ASCII format, making it possible to read and edit them. However, the majority of files are written in the non-editable binary form for preserving the calculations and making them transportable between different computer systems. The most important generated binary files during a model reduction are

- `.sub` : The superelement matrix file, which contains the reduced matrices $\mathbf{m} = \mathbf{H}^T\mathbf{M}\mathbf{H}$, $\mathbf{k} = \mathbf{H}^T\mathbf{K}\mathbf{H}$ and load vectors if any load is applied during the generation pass.
- `.emat` : Element matrices file containing matrices for all elements in the model.
- `.esav` : Element saved data files
- `.seld` : Superelement load vector data from generation pass.
- `.ln22` : Factorized stiffness matrix from the sparse solver
- `.full` : Assembled global stiffness and mass matrices for the full model.
- `.db` : Database file.

Among these, the `.sub` file is of particular interest, because it contains the mass and stiffness matrix for the reduced system. At a later stage, this file will be read and the

matrices converted and written out in ASCII format.

The following APDL code snippet shows a small example of a generation pass implemented in APDL code. It shows how one can define variables, as well as modelling a line, meshing it and selecting the element type to be used for the analysis.

Example of a generation pass in APDL

```
1  !
2  ! Example of a generation pass in APDL
3  !
4
5  / title , Cantilever Beam
6  /filname, gen                    ! Changing filename for clarity
7  save                    ! Saving
8
9  ! Defining parameters
10 *set,youngs,20.58e10 ! Young's modulus [N/m^2]
11 *set, density,7800                ! Density [kg/m^3]
12 *set, length, 0.6                 ! Length [m]
13 *set, nElements, 10               ! Number of elements used
14 /prep7
15 k,1,0,0              ! Enter Keypoints
16 k,2, length,0
17 l,1,2                     ! Create Line between keypoints
18 et,2, beam188                  ! Element Type for non−super elements
19 mp,ex,1,youngs            ! Defining Young's Modulus
20 mp,prxy,1,0.33                  ! Poisson's Ratio
21 mp,dens,1, density                       ! Density
22 sectype,1, beam, rect, , 0          ! Setting the beam's cross section
23 secoffset, cent            ! Setting section offset
24 secdata, 0.08, 0.005,0,0,0,0,0,0,0,0,0,0          ! Defining cross−section
25 lesize, all ,,, nElements           ! Number of elements
26 lmesh, all, , nElements                          ! Mesh Line
27
28 /solu
29 antype, substr    ! Defining a substructuring / CMS analyis type.
30 seopt,gen,2,0,0,0            ! Defining options for substructuring analysis. Generating K
       and M matrices
31 lumpm,0            ! Not using lumped mass matrix.
32 m, 1, all          ! Selecting node 1 and 2 as master nodes. "Giving them access" to all
       degrees of freedom.
33 m, 2, all
34 save
35 solve                            ! Issuing the solve command
36 finish
37 /quit
```

The Use Pass

In the use pass, the superelement is used in analyses. Either by being a part of the model, connected to other non-superelements, or the entire model may be a superelement. Any analysis type except explicit dynamics analysis could be performed in the use pass, just like any other analysis. The difference being that yourm model now contains the superelement generated in the generation pass. The superelement portion of the model needs to be defined with the ANSYS element type MATRIX50 for the solver to know it is a super element. When the `solve` command is issued, the results will consist of a complete solution for the non-superelements, and a reduced solution containing the solution just at the selected master nodes, for the super elements. The reduced solution will be written to the file `use.dsub`.

The Expansion Pass

In the expansion pass, the reduced solution `use.dsub` is expanded, and results at all degrees of freedom in the superelement is calculated. Multiple expansion passes could be issued if multiple superelements were created in the use pass. The solver uses the files `gen.asav`, `gen.full`, `gen.sub`, `gen.ln22`, `gen.db` and `gen.seld` from the generation pass and `use.dsub` from the use pass. The type of expansion pass method is automatically detected by the solver.

2.3.3 Accuracy of Model Reduction Performed in ANSYS

The Mechanical APDL program has an extensive database of verification test cases used for quality assurance of its element types and solution algorithms. The test cases are based on validated results from published work, and are used for validating new versions of the program with extended functionality. A full overview of available test cases can be seen in [4].

Multiple test cases are available for testing the model reduction methods of Mechanical APDL. This section will present one of them, namely an automotive suspension system seen in Figure 2.6. The model is used for demonstrating the benefits of CMS reduction. Three super elements are created; left and right wheel and the main frame. The main frame is then constrained in the top bolts. Next, a modal analysis is performed, and the first 100 eigenfrequencies are extracted.

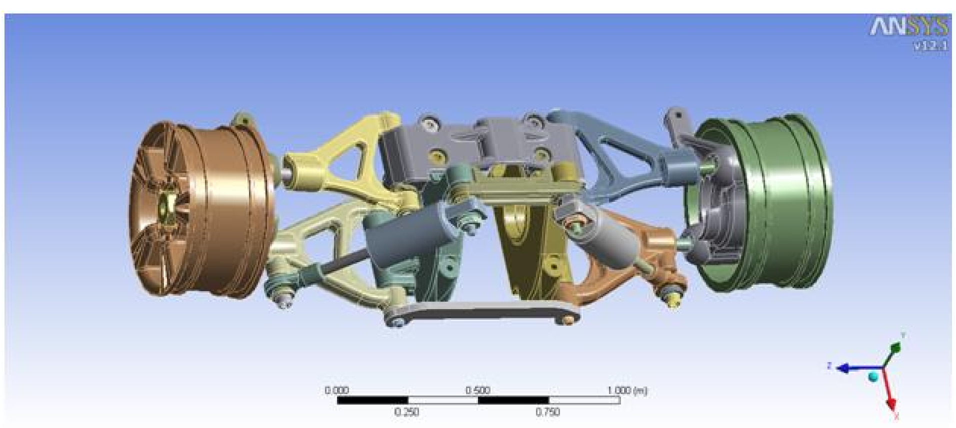

Figure 2.6: Automotive suspension system used for testing CMS reduction accuracy

Mode #	Full Model [Hz]	CMS Model [Hz]	% Diff
1	31.12	31.31	0.59
2	33.13	33.12	0.03
3	44.66	44.69	0.08
4	53.17	53.10	0.12
5	87.88	88.18	0.34

Table 2.1: Comparison of found eigenfrequencies for the automotive suspension assembly in the full model vs the CMS reduced model

An excerpt of the results is presented in Table 2.1, where the resulting 5 first eigenfrequencies for the corresponding modes are shown. It shows a very little loss of accuracy - a maximum of 0.59% - between analysing the full model and the model created by super elements. The full results could be seen in [5].

Full Model [s]	CMS Model	
	Generation Pass [s]	Use + Expansion Pass [s]
1291.0	1336.0	3.0

Table 2.2: Elapsed time for solving the full model and applying CMS reduction. Note that the generation pass is only necessary to do once when a geometry is set.

The improvement in solution time is also significant when using CMS reduction, as can be seen in Table 2.2, where the elapsed time for the modal analysis to calculate the first 100 frequencies of the model is shown. The results show that creating the super elements in the generation pass is a relatively expensive process, with an elapsed time close to solving the

full model. However, once the super elements are created, solving the modal analysis in the use and expansion passes is over 400 times faster than solving the full model. Applying CMS reduction to finite element models can therefore be a major advantage, as it enables a more agile development process, where multiple analyses could be performed rapidly, with a high accuracy for one design.

2.3.4 ANSYS ACT

As model reduction is currently not available through the ANSYS Mechanical application, one needs to create a custom way of accessing the model reduction techniques of APDL through the graphical user interface of ANSYS Mechanical. ANSYS Customization Toolkit (ACT) provides this possibility by enabling users to create apps for the ANSYS environment that could be tailored to fit their engineering problem. ACT can currently (Version 19.2) be used to customize the following ANSYS products:

- AIM
- DesignModeler
- DesignExplorer
- Electronics Desktop
- Fluent
- Mechanical
- SpaceClaim
- Workbench

2.3.5 Structure of an ACT Application

Two basic parts make up an ACT extension:

- An **XML file** defining context, custom GUIs and callbacks to functions.
- An **IronPython script** that contains functions responding to user interactions and GUI events, as well as the app's behavior. The IronPython language is an open-source implementation of the Python programming language. Its strength is its tight integration with the .NET Framework. Using IronPython, the user can use both the .NET Framework and Python libraries. In addition, other .NET languages can use the IronPython code. Further documentation can be seen in [6]

The extension may also contain components like external Python libraries, input files, and images to be displayed in the app. Once the extension is developed, it can be shared in two different formats, seen in Figure 2.7.

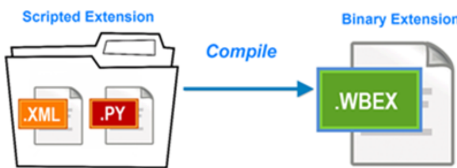

Figure 2.7: The two formats used for sharing ACT extensions: scripted or binary. [1]

The **scripted extension** contains the XML and IronPython script, as well as subdirectories containing additional code, images and image files. The contents of the scripted extension is all editable, allowing easy modification of functions in the extension.

The **binary extension** is a binary WBEX (WorkBench Extension) file that is generated when the scripted extension is built and compiled. The file could easily be shared and installed on another computer with an ANSYS license. The contents of the extension can not be edited, meaning that a new version of the WBEX file needs to be built, compiled and shared for the users to see changes made in the extension.

For the ANSYS Mechanical program, ANSYS ACT offers a wide range of possibilities for customisation:

- APDL macros can be run in the background, opening for automation of repetetive solver steps
- Having access to APDL scripting through ANSYS Mechanical enables the user to use solver capabilities of APDL that are not exposed in ANSYS Mechanical.
- Pre-processing features can be added, like custom loads and boundary conditions.
- Post-processing features could be developed in order to show custom results specifically tailored for the problem to solve.
- Third-party solvers are possible to integrate in the ANSYS environment for solver customisation.
- Graphics in ANSYS Mechanical is possible to tailor, opening possibilities to display custom information like drawing lines, surfaces and text descriptions.

2.3.6 ANSYS Element Library

The ANSYS element library contains all possible element types ANSYS offers for finite element analysis. Currently (Version 19.2) it contains 146 different elements, each specialized for its intended use. Elements have the following characteristics:

[1] Image courtesy of [7].

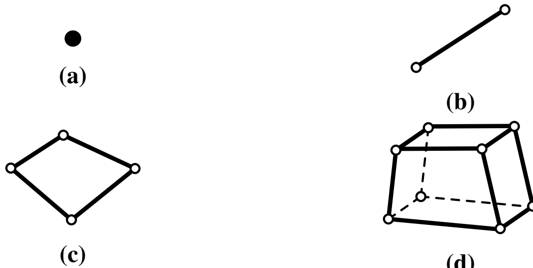

Figure 2.8: The four main element types: (a) Mass element, (b) Line element, (c) Area element, (d) Volume element. Element illustrations taken from [8]

- **Element name**: All element types have a unique name on a maximum of eight characters such as BEAM188. It consists of a group label (BEAM) as well as a unique number (188).
- **2D/3D**: The model is set in 2D or 3D depending on what elements to use. 2D models are defined in the X-Y plane and run faster than equivalent 3D models. The model becomes 3D if any 3D elements are used. A 2D element may be used in a 3D model.
- **Element Shape**: All elements are in general divided into four shapes, seen in Figure 2.8:
  - Point elements are defined by one node. Point elements are typically mass elements.
  - Line elements are represented by a line or arc that is connecting two or three nodes. Typical elements are beams, pipes and axisymmetric shells.
  - Area elements have triangular or quadrilateral shape. It might be a 2D solid element or a shell element.
  - Volume elements have tetrahedral or brick shape. Typical elements are 3D solid elements.
- Discipline: Specialized elements for the following disciplines are available:
  - Structural
  - Thermal
  - Acoustic
  - Diffusion
  - Fluid
  - Magnetic Electric
  - Electric Circuit
  - Coupled-Field
  - Contact
  - Combination
  - Matrix
  - Infinite
  - Load
  - Meshing
  - Reinforcing
  - User-defined

SHELL181

The SHELL181 element (Figure 2.9 is a 4-node (I, J, K, L) strucural shell element for 3D space. It has six degrees of freedom at each node: Translation in x, y, z and rotation about the x, y, z axis. It is suitable for analyzing thin to moderately-thick shell structures. Its shell thickness needs to be defined with the APDL commands

Commands for Setting Thickness at SHELL181 Element

```
1  setcype, , shell
2  secdata, thickness
```

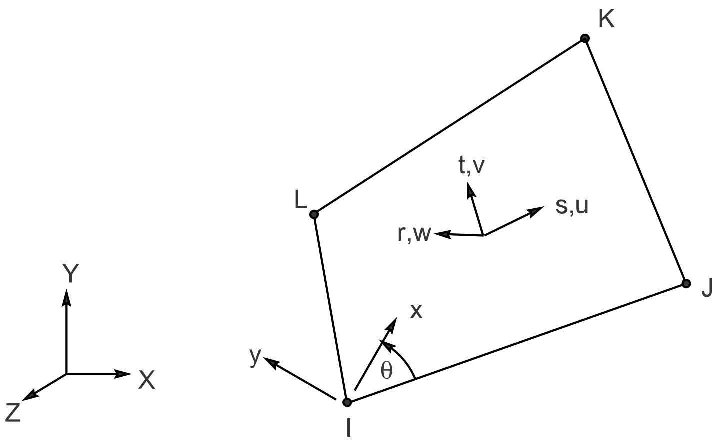

Figure 2.9: The SHELL181 element from the ANSYS element library [9]

For defining the shape functions of SHELL181, the following definitions and notations are used, referencing Figure 2.9: [8]

- As this is a shell element, the element coordinate system is not the same as the global Cartesian system. For SHELL181, u and v are in-plane motions and w is out-of-plane motion.
- Coordinates s, t and r are normalized, going from -1.0 on one side of the element, to $+1.0$ on the other side. They are not necessarily orthogonal to each other.
- Subscripted variables such as u_J refer to the u motion at node J.

For the stiffness and mass matrix, the following shape functions are used:

$$\begin{aligned} u = \frac{1}{4}(&u_I(1-s)(1-t) + u_J(1+s)(1-t) \\ &+ u_K(1+s)(1+t) + u_L(1-s)(1+t)) \end{aligned} \tag{2.35}$$

$$v = \frac{1}{4}(v_I(1-s)(1-t) + \ldots \text{(analogous to } u) \tag{2.36}$$

$$w = \frac{1}{4}(w_I(1-s)(1-t) + \ldots \text{(analogous to } u) \tag{2.37}$$

$$\begin{aligned}\theta_x = \frac{1}{4}(&\theta_{x,I}(1-s)(1-t) + \theta_{x,J}(1+s)(1-t) \\ &+ \theta_{x,K}(1+s)(1+t) + \theta_{x,L}(1-s)(1+t))\end{aligned} \tag{2.38}$$

$$\theta_y = \frac{1}{4}(\theta_{y,I}(1-s)(1-t) + \ldots \text{(analogous to } \theta_x) \tag{2.39}$$

$$\theta_z = \frac{1}{4}(\theta_{z,I}(1-s)(1-t) + \ldots \text{(analogous to } \theta_x) \tag{2.40}$$

If a lumped mass approximation is selected, only (2.35) - (2.37) is used.

2.3.7 Degree of Freedom Ordering

ANSYS use an internal ordering method for the set of degrees of freedom (DOFs). When assembling the full mass and stiffness matrix $\mathbf{M}, \mathbf{K}$, ANSYS saves matrix data in a binary file located in the solver directory with the extension `.FULL`. Items saved in the `.FULL` file uses an **internal** solver ordering. The ordering is obtained by an internal ANSYS algorithm that reorders the equations in order to minimize the solver time and disk requirements. The DOF ordering is specified in the file `filename.mapping`, written out by setting the `Mapping` parameter to *Yes* in the APDL command `HBMAT`. An example of the `.mapping` file for the simple *twoQUAD4* model seen in Figure 4.1 is shown here.

Mapping file for the twoQUAD4 model

Matrix Eqn	Node	DOF
1	5	UX
2	5	UY
3	5	UZ
4	5	ROTX
5	5	ROTY
6	5	ROTZ
7	4	UX
8	4	UY
9	4	UZ
10	4	ROTX
11	4	ROTY
12	4	ROTZ
13	1	UX
14	1	UY
15	1	UZ
16	1	ROTX
17	1	ROTY

18	1	ROTZ
19	2	UX
20	2	UY
21	2	UZ
22	2	ROTX
23	2	ROTY
24	2	ROTZ
25	6	UX
26	6	UY
27	6	UZ
28	6	ROTX
29	6	ROTY
30	6	ROTZ
31	3	UX
32	3	UY
33	3	UZ
34	3	ROTX
35	3	ROTY
36	3	ROTZ

2.4 Fedem

Fedem, an acronym for Finite Element Dynamics in Elastic Mechanisms, is a computer program that provides features for creating, solving and post-processing a model in a 3D graphical environment. The program has multiple solver modules for performing different types of calculations. Detailed descriptions of each module can be seen in [10] (Fedem Version 7.2), and are briefly discussed here:

- **Reducer**: Performs a CMS reduction of the mass and stiffness matrices of a FE model for faster simulation of nonlinear dynamics.
- **Dynamics Solver**: Performs a non-linear dynamics simulation of the superelements' reaction over time to displacements and control system output.
- **Stress Recovery**: Recovers stresses and strain in the internal nodes from the deformations of the external nodes imposed at the model.
- **Mode Shape Recovery**: Recovers mode shapes from the eigenvalue results of the Dynamics solver.
- **Strain Rosette Analysis**: Applies virtual strain gauges on the FE model and outputs strain and stresses from the model over time similar to output from real strain gauges.
- **Strain Coat Analysis**: Recovers stresses and strains from the coat elements in a FE model, outputting the recovered stresses and strains for the model over the entire time history.

- **Curve Export Utility**: Allows the user to automatically export result curves to a single ASCII file.

2.4.1 Model Reduction in Fedem

The Fedem Reducer has implemented the Guyan reduction and CMS reduction techniques presented in Section 2.2 for effectively solving dynamic problems. The reduction technique is well suited for flexible mechanism analyses due to it preserving the effective masses and inertias of the model. The external nodes needed for the reduction are defined during modelling of the mechanism as ”triads”. The triads are defined on the connection points for joints, springs, dampers, external loads external load, control inputs or other points of interest. The reduction begins automatically when a dynamics simulation is started, and Fedem determines which parts that need to be reduced based on how the triads are chosen and their connection to the rest of the model. The number of component modes to use for the reduction is specified by the user before reduction. It is recommended to include the lowest modes of vibration in order to achieve good results.

2.4.2 Importing reduced models to Fedem

As Fedem does not support modelling or meshing of finite element models, they need to be created in external CAE systems, stored in separate files and imported into Fedem. Fedem supports importing files using the Nastran Bulk Data Format (`.nas` or `.bdf`), SESAM Input Interface File Format (`.fem`) as well as the older Fedem Link Model format (`.flm`). After reading these files, Fedem stores the info retrieved in the Fedem Technology Link Format (`.ftl`).

The `.ftl` file format contains all data needed for defining FE parts. It is defined in ASCII format, and can thus easily be edited using a text editor. The file contains a set of identifiers and parameters expressed with the same syntax:

```
identifier{id value1 value2 ...  valueN {reference id text}}
```

where the parameters are listed in Table 2.3 taken from [10]

An example of an identifier with attributes is

```
QUAD4{4 22 34 12 32{PMAT 1}}
```

This defines a 4-noded tetrahedron element with ID=4 that is coupled to the nodes 22, 34, 12 and 32. The element uses an attribute of type PMAT with ID=1.

```
PMAT{1 2.10e+11 8.00e+10 2.90e-01 7.82e+03}
```

This defines the material property that is referred to in the QUAD4 element. The decimal numbers describe material parameters like Young's modulus, shear modulus, Poisson's ratio and density. A comprehensive guide of the available identifiers can be found in [10, p. 298]

Name	Description
*identifier**	Specifies field type (e.g., element type, attribute type).
*id**	Unique ID for the field (relative to the other fields with the same identifier).
value1 … valueN	Primary values for the object (can be text, integers, or decimal digits).
references	Additional data or other fields can be referred to using this field.
reference and id	Field reference (reference specified in combination with a valid ID).
text	Can be used as additional information for a field reference or as an optional tag (e.g., a group name)

Table 2.3: Syntax of Technology Link Format

In order to perform calculations on imported models, Fedem does also need matrix files from the reduced models. Those are saved internally as binary `.fmx` files after model reduction is performed. Binary `.fmx` files are created for the reduced stiffness and mass matrices, as well as for the gravity vector described in Section 2.4.3

2.4.3 Gravity Vectors in Fedem

Gravitational forces in Fedem are calculated from unit gravitational acceleration vectors, as described in [11, p. 118]. Unit acceleration in the x, y and z direction is denoted $\mathbf{U}_x$, $\mathbf{U}_y$ and $\mathbf{U}_z$ respectively. The unit vectors are constructed so that for all degrees of freedom in $\mathbf{U}_x$ that correspond to x translation, the acceleration component of $\mathbf{U}_x$ is set to 1, otherwise 0. $\mathbf{U}_x$ will be of size $((n_e + n_i) \cdot DOFs) \times 1)$ where n_e is the number of external nodes, and n_i is the number of internal nodes. The total number of nodes in the model is $n = n_e + n_i$. For an element with 6 DOFs per node, $\mathbf{U}_x$, $\mathbf{U}_y$ and $\mathbf{U}_z$ will then be

$$\mathbf{U}_x = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0\end{bmatrix}^T$$
$$\mathbf{U}_y = \begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \cdots & 0\end{bmatrix}^T$$
$$\mathbf{U}_z = \begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0\end{bmatrix}^T$$

The gravitational forces $\mathbf{G}_x$ corresponding to $\mathbf{U}_x$ are calculated from

$$\begin{bmatrix}\mathbf{G}_{xe} \\ \mathbf{G}_{xi}\end{bmatrix} = \begin{bmatrix}\mathbf{M}_{ee} & \mathbf{M}_{ei} \\ \mathbf{M}_{ie} & \mathbf{M}_{ii}\end{bmatrix} \begin{bmatrix}\mathbf{U}_{xe} \\ \mathbf{U}_{xi}\end{bmatrix} = \begin{bmatrix}\mathbf{M}_{ee}\mathbf{U}_{xe} + \mathbf{M}_{ei}\mathbf{U}_{xi} \\ \mathbf{M}_{ie}\mathbf{U}_{xe} + \mathbf{M}_{ii}\mathbf{U}_{xi}\end{bmatrix} \tag{2.41}$$

The forces are then reduced by the CMS tranformation matrix $\mathbf{H}$ to $\mathbf{g}_x$:

$$\begin{bmatrix}\mathbf{g}_{xe}\\ \mathbf{g}_{xg}\end{bmatrix} = \begin{bmatrix}\mathbf{I} & \mathbf{B}^T\\ \mathbf{0} & \mathbf{\Phi}\end{bmatrix}\begin{bmatrix}\mathbf{G}_{xe}\\ \mathbf{G}_{xi}\end{bmatrix} = \begin{bmatrix}\mathbf{G}_{xe} + \mathbf{B}^T\mathbf{G}_{xi}\\ \mathbf{\Phi}^T\mathbf{G}_{xi}\end{bmatrix} \tag{2.42}$$

When inserting (2.41), and taking into account the symmetry property

$$\mathbf{M}_{ei} = \mathbf{M}_{ie}^T \tag{2.43}$$

the full expression for $\mathbf{g}_{xe}$ and $\mathbf{g}_{xg}$ becomes

$$\begin{aligned}\mathbf{g}_{xe} &= \mathbf{M}_{ie}^T\mathbf{U}_{xi} + \mathbf{M}_{ee}\mathbf{U}_{xe} + \mathbf{B}^T\mathbf{M}_{ii}\mathbf{U}_{xi} + \mathbf{B}^T\mathbf{M}_{ie}\mathbf{U}_{xe}\\ \mathbf{g}_{xg} &= \mathbf{\Phi}^T\mathbf{M}_{ii}\mathbf{U}_{xi} + \mathbf{\Phi}^T\mathbf{M}_{ie}\mathbf{U}_{xe}\end{aligned} \tag{2.44}$$

Finding gravitational forces in the y and z direction is an equivalent process by change of indexes:

$$\begin{aligned}\mathbf{g}_{ye} &= \mathbf{M}_{ie}^T\mathbf{U}_{yi} + \mathbf{M}_{ee}\mathbf{U}_{ye} + \mathbf{B}^T\mathbf{M}_{ii}\mathbf{U}_{yi} + \mathbf{B}^T\mathbf{M}_{ie}\mathbf{U}_{ye}\\ \mathbf{g}_{yg} &= \mathbf{\Phi}^T\mathbf{M}_{ii}\mathbf{U}_{yi} + \mathbf{\Phi}^T\mathbf{M}_{ie}\mathbf{U}_{ye}\end{aligned} \tag{2.45}$$

$$\begin{aligned}\mathbf{g}_{ze} &= \mathbf{M}_{ie}^T\mathbf{U}_{zi} + \mathbf{M}_{ee}\mathbf{U}_{ze} + \mathbf{B}^T\mathbf{M}_{ii}\mathbf{U}_{zi} + \mathbf{B}^T\mathbf{M}_{ie}\mathbf{U}_{ze}\\ \mathbf{g}_{zg} &= \mathbf{\Phi}^T\mathbf{M}_{ii}\mathbf{U}_{zi} + \mathbf{\Phi}^T\mathbf{M}_{ie}\mathbf{U}_{ze}\end{aligned} \tag{2.46}$$

The matrix used by Fedem for calculating gravitational forces is then the assembled matrix $\mathbf{G}$:

$$\mathbf{G} = \begin{bmatrix}\mathbf{g}_{xe} & \mathbf{g}_{ye} & \mathbf{g}_{ze}\\ \mathbf{g}_{xg} & \mathbf{g}_{yg} & \mathbf{g}_{zg}\end{bmatrix} \tag{2.47}$$

$\mathbf{G}$ will be of size $((p \cdot DOFs + s) \times 3)$ where p is the number of external nodes, $DOFs$ is the number of degrees of freedom per node and s is the number of selected component modes.

Calculating the Gravity Vector from the Reduced Mass Matrix

The gravity vectors in Fedem could also be calculated from the reduced mass matrix $\mathbf{m} = \mathbf{H}^T\mathbf{M}\mathbf{H}$. The gravity vectors could then simply be calculated as

$$\begin{bmatrix}\mathbf{g}_{xe}\\ \mathbf{g}_{xg}\end{bmatrix} = \begin{bmatrix}\mathbf{m}_{11} & \mathbf{m}_{12}\\ \mathbf{m}_{21} & \mathbf{m}_{22}\end{bmatrix}\begin{bmatrix}\mathbf{U}_{xe}\\ \mathbf{U}_{xc}\end{bmatrix} = \begin{bmatrix}\mathbf{m}_{11}\mathbf{u}_{xi} + \mathbf{m}_{12}\mathbf{u}_{xe}\\ \mathbf{m}_{21}\mathbf{u}_{xi} + \mathbf{m}_{22}\mathbf{u}_{xe}\end{bmatrix} \tag{2.48}$$

The displacement vector $\mathbf{U}_{xe}$ is the same as described in Section 2.4.3. $\mathbf{U}_{xc}$ is of size $(s \times 1)$ and is set to 0. Expanding (2.48) gives for all directions:

$$\begin{aligned}\mathbf{g}_x &= \mathbf{H}^T\mathbf{M}\mathbf{H}\mathbf{U}_x^{\text{red}}\\ \mathbf{g}_y &= \mathbf{H}^T\mathbf{M}\mathbf{H}\mathbf{U}_y^{\text{red}}\\ \mathbf{g}_z &= \mathbf{H}^T\mathbf{M}\mathbf{H}\mathbf{U}_z^{\text{red}}\end{aligned} \tag{2.49}$$

Comparing this to what is given when using the full mass matrix, we see that gravity vectors from the full mass matrix are given by

$$\begin{aligned} \mathbf{g}_x^{\text{full}} &= \mathbf{H}^T \mathbf{M} \mathbf{U}_x^{\text{full}} \\ \mathbf{g}_y^{\text{full}} &= \mathbf{H}^T \mathbf{M} \mathbf{U}_y^{\text{full}} \\ \mathbf{g}_z^{\text{full}} &= \mathbf{H}^T \mathbf{M} \mathbf{U}_z^{\text{full}} \end{aligned} \tag{2.50}$$

where $\mathbf{U}_i^{full}$ are defined as in Section 2.4.3.

2.4.4 The FFQ4 Shell Element in Fedem

FFQ4, seen in Figure 2.10, is a 4-node quadrilateral shell element used for modelling shell structures in Fedem. It is composed of a Quadrilateral plate Bending Element with Shear deformation (QBESH) and a Quadrilateral Membrane element with Rotational degrees of Freedom (QMRF). The element nodes are numbered clockwise 1-2-3-4, referring to Figure 2.10. Each of the four nodes have six degrees of freedom: u, v, w, r_x, r_y, r_z where r_i refer to rotation about axis i.

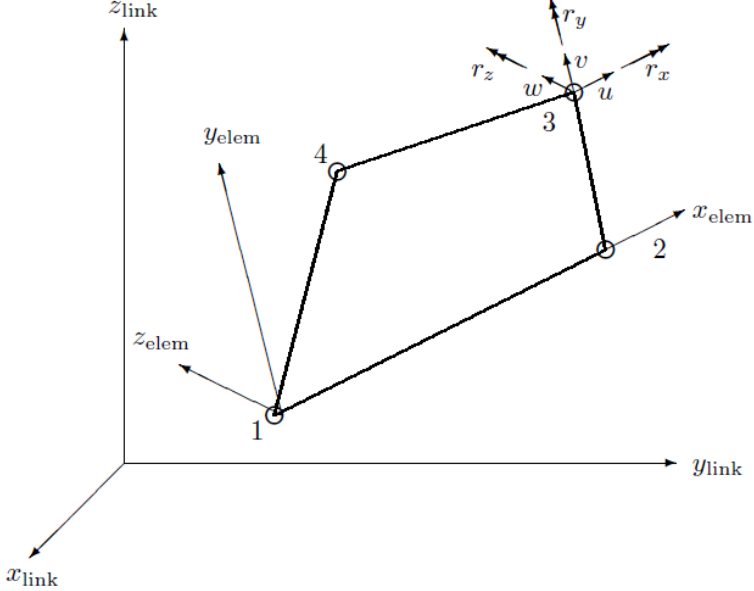

Figure 2.10: FFQ4 element used in Fedem [2]

The FFQ4 shell element is unfortunately poorly documented, but is based on free formulation theory developed by Pål Bergan, Magne Nygård et.al. where an element is constructed

[2] Image courtesy of [12]

of two parts: one for describing rigid body modes, and one describing higher order terms [13, 14]. From Fedem version 7.3, the FFQ4 shell element is replaced with a new element: ANDES, proposed by Felippa, Militello in [15] and researched by Haugen, Skallerup in [16].

2.4.5 Degree of Freedom Ordering in Fedem

The mass and stiffness matrix in Fedem is, similarly to ANSYS, restructured by the Fedem solver, in order to reduce disk requirements and make the solving process more efficient. The mapping of equation number to corresponding degree of freedom is defined in the file `MEQN.res`, which is written to the Fedem solver directory by the additional solver option `-debug 3`. This mapping is used in the CMS reduction process, when the full mass matrix is partitioned into

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}_{ee} & \mathbf{M}_{ei} \\ \mathbf{M}_{ie} & \mathbf{M}_{ii} \end{bmatrix}$$

The partitioning process is easier understood by a small example. We create a custom mass matrix where every element in the matrix is set to its corresponding index number (row-column):

$$\mathbf{M} = \begin{bmatrix} 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \\ 31 & 32 & 33 & 34 & 35 \\ 41 & 42 & 43 & 44 & 45 \\ 51 & 52 & 53 & 54 & 55 \end{bmatrix}$$

The mapping of degree of freedom to the corresponding equation is given by the following Python-style dictionary:

```
1  mapping = {
2      1: 3,
3      2: 4,
4      3: 5,
5      4: 1,
6      5: 2
7  }
```

The dictionary maps DOF number to the corresponding equation: `mapping[dof] = equation`, e.g. `mapping[1] = 3`, `mapping[4] = 1`.

When the matrix is partitioned, the equations corresponding to the external nodes are moved to $\mathbf{M}_{ee}$, while the equations corresponding to the internal nodes are moved to the $\mathbf{M}_{ii}$ part of the matrix. The $\mathbf{M}_{ei}$, $\mathbf{M}_{ie}$ parts of the matrix will be a combination of the equations from the external and internal nodes. In addition, Fedem sorts the partitioned parts by **ascending equation number**.

The partitioning process will first move the rows corresponding to the equation numbers of the external and internal degrees of freedom. If the external degrees of freedom are chosen to be $\begin{bmatrix} 2 & 5 \end{bmatrix}$ and internal degrees of freedom are $\begin{bmatrix} 1 & 3 & 4 \end{bmatrix}$, the $\mathbf{M}$ matrix will be restructured to the following after moving the **rows**:

$$\begin{bmatrix} 21 & 22 & 23 & 24 & 25 \\ 41 & 42 & 43 & 44 & 45 \\ 11 & 12 & 13 & 14 & 15 \\ 31 & 32 & 33 & 34 & 35 \\ 51 & 52 & 53 & 54 & 55 \end{bmatrix}$$

Next, the **columns** are moved, resulting in the partitioned matrix

$$\begin{bmatrix} 22 & 24 & 21 & 23 & 25 \\ 42 & 44 & 41 & 43 & 45 \\ 12 & 14 & 11 & 13 & 15 \\ 32 & 34 & 31 & 33 & 35 \\ 52 & 54 & 51 & 53 & 55 \end{bmatrix}$$

The partitions of the matrix will then be

$$\mathbf{M}_{ee} = \begin{bmatrix} 22 & 24 \\ 42 & 44 \end{bmatrix} \quad \mathbf{M}_{ei} = \begin{bmatrix} 21 & 23 & 25 \\ 41 & 43 & 45 \end{bmatrix}$$
$$\mathbf{M}_{ie} = \begin{bmatrix} 12 & 14 \\ 32 & 34 \\ 52 & 54 \end{bmatrix} \quad \mathbf{M}_{ii} = \begin{bmatrix} 11 & 13 & 15 \\ 31 & 33 & 35 \\ 51 & 53 & 55 \end{bmatrix}$$

2.5 Software Development

When developing software, many related processes lead to the production of a software system; specifications for the software's functionality are set, the software is developed and it is validated to ensure it meets the set specifications. Hence, the process of how one develops software should be investigated in order to optimize the process. Multiple models for software processes exist. For simplification, it is possible to divide the type of processes in two: plan-driven processes and agile processes. In **plan-driven processes** all process activities are planned in advance of the development. Progress made in the project as well as the resulting product is then measured against the plans set before development has started. In **agile processes**, the planning is continuous, and is a subject of change through the whole development process. Incremental development is one type of agile process.

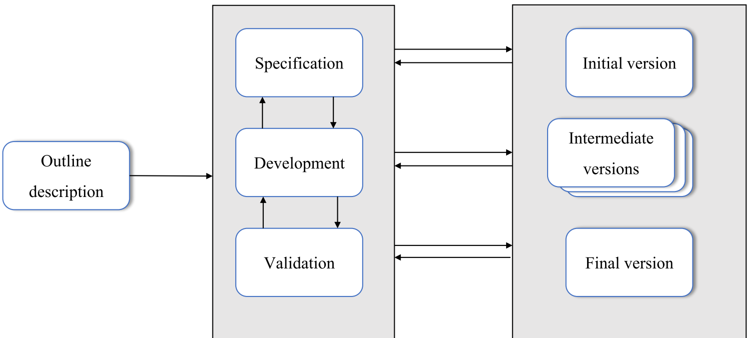

Figure 2.11: Incremental development process.

When software is developed incrementally, it is based on the idea that one develops an initial version of the software. This is then tested and feedback is generated. The software will then evolve in incremental versions until a system meeting the requirements set has been developed. The specification, development and validation phases of the project will always be subject for change through the whole development. This is illustrated in Figure 2.11. In this way it reflects the way humans solve problems: a complete solution to a problem is rarely known in advance, but we move closer to a solution in small steps, and backtrack if a mistake is discovered. Sommerville [17] states three major advantages to incremental development:

- There is little cost related to implementing changes in requirements, as the specification, development and validation processes are always continuous and open for change. This means the amount of analysis and documentation that has to be redone when requirements change is reduced.
- It is easier to get feedback on work in development, as one at an early stage of the development process tries to have a working, initial version of the product. For people outside the development process, it is easier to judge a product and perhaps edit its specifications if they can have it demonstrated.
- It is possible to gain value from a product at an early stage in the process, as some stand-alone features might be available for use.

2.5.1 Validating Software by Unit Testing

Unit testing is the process of testing the small units that make up a computer program. Individual functions are usually seen as the simplest type of component, and is therefore usually the target for unit testing. When performing unit testing, the aim is simply to test if your functions behave as expected. This is accomplished by calling these routines with a range of different input parameters, and montoring if the expected result is returned or produced. Ideally, functions should be tested in isolation. This means that the test should

be free of dependencies to other parts of the program. In this way, we ensure that we are actually testing the desired unit of the program.

IronPython offers an automated unit testing framework `unittest` [18] for writing and running tests. It gives the possibility to run all tests written for a program, and reports the outcome of each individual test. The automated tests of `unittest` consist of three parts:

1. **Setup part**: Here, one can set up the initial state of the program. The setup is run before each single test, ensuring equal initial conditions for each test.
2. **Call part**: This is where the object or function to test is being called from the testing framework.
3. **Assertion part**: The behavior of the function is compared to what is expected. An evalutaion of the assertion that evaluates to `True` means that the test has been successful, if it is `False`, it has failed.

The following example, taken from [18] with small modifications, shows a small, basic example of testing the behavior of Python's `random` module

Basic Unit Test Example

```
1  import random
2  import unittest
3
4  class TestSequenceFunctions(unittest.TestCase):
5
6      def setUp(self):
7          self.seq = range(10)
8
9      def test_shuffle(self):
10         # make sure the shuffled sequence does not lose any elements
11         random.shuffle(self.seq)
12         self.seq.sort()
13         self.assertEqual(self.seq, range(10))
14
15     def test_choice(self):
16         element = random.choice(self.seq)
17         self.assertTrue(element in self.seq)
18
19 if __name__ == '__main__':
20     unittest.main()
```

Chapter 3

Method

ANSYS supports the same CMS model reduction as implemented in Fedem. This means that ANSYS is able to generate the same reduced mass, stiffness and transformation matrix $\mathbf{H}$ as needed by Fedem to perform dynamics analyses. And this again means that the reduced models from ANSYS could be used in the Fedem environment, allowing the following benefits:

- Creating models could be performed in ANSYS with the CAD programs ANSYS SpaceClaim or ANSYS DesignModeler. The CAD programs support parametrization of the models, which in return facilitates the analysis of design changes.
- The ANSYS Meshing program could be used to create finite element meshes. The program supports advanced mesh configuration tools like automated inflation, physics-aware meshing and controls for moving, merging and editing nodes and elements.
- When meshing is performed in ANSYS, the full finite element library [9] of ANSYS is available. Currently (Version 19.2) it contains 146 different elements. Each element type is thoroughly documented, and specific elements for different applications are available.
- The extensive material data and material designer capabilities of ANSYS could be integrated in the models. This makes it possible to model composite materials, anisotropic material and more in the analysis.
- Using a combination of ANSYS and Fedem for analyses would make it possible to verify results to a larger extent than what is currently possible. Identical analyses could be performed using both systems, thus extending the possibilities for verifying results.

This chapter will introduce the ANSYS ACT extension ”ModRed” that will enable an easy-to-use integration of CMS model reduction performed in ANSYS into Fedem. Its implementation details will be explained, and different use cases will be presented.

3.1 The ModRed Extension for Performing CMS Model Reduction in ANSYS

In order to open and analyze models in Fedem, the following matrices and data is needed:

- The reduced mass matrix by CMS reduction: $\mathbf{m} = \mathbf{H}^T\mathbf{M}\mathbf{H}$
- The reduced stiffness matrix by CMS reduction: $\mathbf{k} = \mathbf{H}^T\mathbf{K}\mathbf{H}$
- A FE mesh in `.nas`, `.fem` or `.ftl` format
- The assembled gravity vectors $\mathbf{G}$

The ACT extension ”ModRed” is able to create all of these matrices and data, enabling easier integration between ANSYS and Fedem.

3.1.1 Installation

The ModRed extension will be available both as a compiled `WBEX` file, as well as a scripted extension containing the full source code. Installing the extension is an easy procedure, but differs slightly depending on what type is to be installed:

Compiled WBEX extension
In ANSYS Workbench, go to

Extensions $\rightarrow$ *Install Extension*

and select the `.wbex` file in the file dialog. This will install the extension to the ANSYS environment. If the extension does not show, try checking the ”Loaded” box for ModRed under *Extensions* $\rightarrow$ *Manage Extensions*

Scripted extension
In ANSYS Workbench, open the ACT Start Page tab. From there, go to *Manage Extensions*, click the *Settings* icon and select *Add Folder*. Then select the folder containing the file `ModRed.xml`. This will install the extension into the environment, and also allows editing the source code while using the extension. When opening the ANSYS Mechanical program, the ModRed extension is added to the toolbar, as seen in Figure 3.1

Figure 3.1: The ModRed extension loaded into the toolbar. The left button places ModRed in the ANSYS Mechanical solution tree. The right button exports Fedem data.

Figure 3.1 shows the ModRed app loaded into the ANSYS Mechanical model tree, as a new ”Solution” type. This allows the extension to edit solver commands as well as performing post-processing manipulation of solver data.

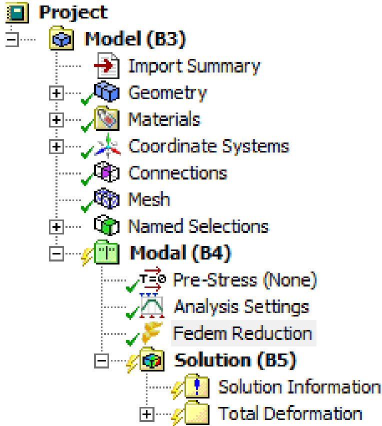

Figure 3.2: The ModRed extension loaded into ANSYS Mechanical

3.1.2 Importing a model into ANSYS Mechanical

There are multiple ways to import models into the ANSYS Mechanical program. Common for all is that this process is handled by ANSYS Workbench.

Importing an Existing Mesh

If the model and mesh is already defined, ANSYS may import the mesh file through the *External Model* module, Figure 3.3

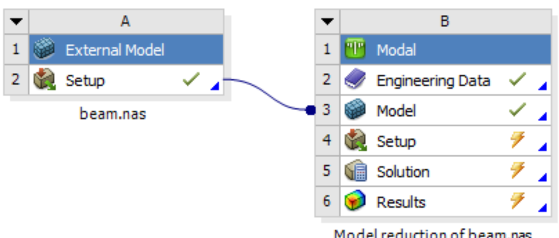

Figure 3.3: Importing an existing mesh into ANSYS Mechanical

From the *External Model* module, the file formats possible to import are listed in Table 3.1

APDL Common Database	Abaqus Input	NASTRAN Bulk Data	Fluent Input	ICEM CFD	LS-DYNA
.cdb	.inp	.bdf, .dat, .nas	.msh, .cas	.uns	.k

Table 3.1: Supported mesh file formats for import to ANSYS Mechanical

When using the *External Model* module to import a meshed model, ANSYS will convert the element type used in the external system to the most similar ANSYS element type in order to be able to solve the system.

Modelling From Scratch in ANSYS

If the modelling is to be performed in ANSYS, a *Geometry* component is used to import the model from the modelling program and into ANSYS Mechanical. Figure 3.4 shows a model made in DesignModeler being imported into ANSYS Mechanical. This method is the most versatile, as the geometry made in ANSYS is possible to parametrize, making it possible to perform analyses on multiple geometrical variations.

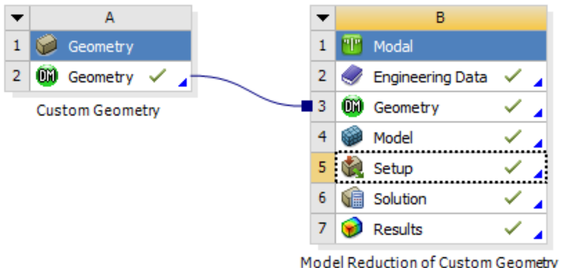

Figure 3.4: Import a Native Model

3.1.3 Defining Settings for the ModRed Extension

Details of "Fedem Reduction"	
Selection of Master Nodes	
Pick Master Nodes By...	Geometry
Scoping Method	Named Selection
Named Selection	eNodes
Model Reduction Method	
Reduction Method	CMS
Num. of Component Modes to Extract	2
File Name	master
Use lumped matrices?	No
Export Options	
Export Matrices to Fedem?	Yes
Fedem File Directory	C:\Users\adrian\Documents\Dok...

Figure 3.5: The ModRed extension loaded into ANSYS Mechanical

Selecting external nodes

As ANSYS is not going to perform any analyses, only calculate the reduced stiffness and mass matrices, one only needs to perform the generation pass of CMS reduction in ANSYS. Here, the user selects what nodes to be used as external or ”triads” in Fedem. ANSYS will then mark these as master nodes when the APDL solver is invoked. The external nodes may be selected in two ways:

Geometry Selection: Using the integrated ANSYS tool for selecting geometry, the user may simply select the nodes that will be set as triads in Fedem. Using the node selection tool, Figure 3.6, changes the view to render the mesh, and allows only nodal points to be selected. Press *Ctrl* while selecting nodes to select multiple nodes.

Figure 3.6: Node selection tool

Named Selection: ANSYS also allows selecting the desired nodes and saving them to a *Named Selection* which may be accessed from the ModRed settings. In this way, the user can select and save multiple sets of external nodes in order to decide what will yield the best results.

Setting Model Reduction Settings

The ModRed extension allows the user to select two types of model reduction: Guyan and CMS reduction. If the *Guyan* reduction option is selected, no additional component modes

are selected, and the transformation matrix is $\mathbf{H} = \mathbf{H}_{\text{Guyan}}$ from Section 2.2.1. If the *CMS* reduction method is chosen, the user also gets the possibility of setting the number of component modes to be extracted from the model. The transformation matrix will then be set as $\mathbf{H} = \mathbf{H}_{\text{CMS}}$, from Section 2.2.2.

The *File name* tab allows the user set a file name that will be given to the `.ftl` file and binary `.fmx` files to be generated. This is useful if multiple parts is to be reduced. A unique name for each part will then make the process of importing the data into Fedem easier.

The user can also select whether or not to use lumped matrices. Setting the option to *Yes* will force the APDL solver to use the `LUMPM` command that makes the solver use a lumped mass approximation for reducing the disk space needed to save the matrices.

Setting Export Options

In the *Export Options*, the user can select whether the matrices are to be exported or not. This is beneficial if the user wants to force ANSYS to not export new matrices on every *Solve* command. In the *Fedem File Directory* tab, the user selects to which folder to output the generated `.ftl` and `.fmx` files.

3.1.4 The APDL Commands Written by ModRed

Once all necessary settings to perform the model reduction has been set, the model is ready to be solved. This is done by hitting the *Solve* button. This will then invoke the `onSolve()` method in `ModRed.py`, which adds APDL commands to the `ds.dat` file to be read by the solver, as explained in Section 2.3.1. The code below shows an excerpt from the `ds.dat` file that is generated when solving a model with nodes 1, 2, 3, 4 selected as the external nodes. See the comments marked with ! for more information about each command. Extensive documentation for each command is found in [19].

Excerpt from ds.dat

```
1  ! ****** Begin Command Snippet ******
2  !
3  ! Commands written by ModRed
4  !
5   finish
6  /filname, master          !Setting the file name
7  save
8  /solu
9  antype, substr   !Defining a substructure analysis (includes both Guyan and CMS)
10 outpr, nsol, all     ! Print command that must be defined if iokey=="tcms"
11 cmsopt, fix ,2,,,,, tcms  !Fixed interface normal modes with 2 component modes
12 seopt, master, 2, 0, 0, ,    ! Super element name and generating mass+stiffness matrix
13 m, 1, all        ! Setting node 1-4 as master nodes, "having access to" all degrees of
       freedom
```

```
14  m, 2,  all
15  m, 3,  all
16  m, 4,  all
17  alls                ! Selecting all nodes before solving
18  solve
19  save
20  finish
21  ! Master node selection finished
22  ! Exporting matrices ...
23  *dmat, cst, d, import, tcms, master.tcms, cst    ! Importing the CST matrix
24  *dmat, nor, d, import, tcms, master.tcms, nor    ! Importing the NOR matrix
25  save
26  *export, cst, mmf, CST.mmf ! Exporting CST and NOR matrices to MMF format
27  *export, nor, mmf, NOR.mmf
28  /aux2
29  fileaux2, master, sub   ! Specifying to dump the reduced matrices
30  hbmat, M_red, hbmat, , ascii, mass, no  !Dumping reduced mass and stiffness matrix to
        Harwell-Boeing format (16-decimal precision)
31  hbmat, K_red, hbmat, , ascii, stiff, no
32  fileaux2, master, full  ! Specifying to dump full matrices
33  hbmat, M_full, hbmat, , ascii, mass, no, yes    ! Dumping full mass matrix in
        Harwell-Boeing format (16-decimal precision)
34  finish
35  *smat, M_red, d, import, hbmat, M_red.hbmat, ascii   ! Importing Harwell-Boeing format
        matrices
36  *smat, M_full, d, import, hbmat, M_full.hbmat, ascii
37  *smat, K_red, d, import, hbmat, K_red.hbmat, ascii
38  *export, M_red, mmf, M_red.mmf      ! Exporting matrices in MMF format (maintains
        16-decimals precision)
39  *export, M_full, mmf, M_full.mmf
40  *export, K_red, mmf, K_red.mmf
41  save
```

3.1.5 Matrix Export

When the model has been solved with the added APDL commands seen in Section 3.1.4, matrices are then exported to the ANSYS solver directory. This can be accessed by right-clicking the *Solution* in the ANSYS Mechanical tree and then selecting *Open Solver Files Directory*. The following matrices are written to this directory:

- `M_full.mmf`: This is the full mass matrix, saved in a sparse format, and formatted in the MatrixMarket file format. Note also the following:
  - The matrix is saved column-first.
  - In order to utilize symmetry and reducing storage space needed, only the lower triangular part of the matrix is saved in the `.mmf` file.

- `M_full.mapping`: Mapping file for the full mass matrix. This relates node to the corresponding equation in the matrix.
- `M_red.mmf`, `K_red.mmf`: The reduced mass and stiffness matrix, saved in sparse format.
- `CST.mmf`: The static part of the transformation matrix $\mathbf{H}$, namely

$$\text{CST} = \begin{bmatrix} \mathbf{I} \\ \mathbf{B} \end{bmatrix} \tag{3.1}$$

  CST is saved in dense form, column-first
- `NOR.mmf`: The component modes part of the transformation matrix $\mathbf{H}$, namely

$$\text{NOR} = \begin{bmatrix} \mathbf{0} \\ \mathbf{\Phi} \end{bmatrix}$$

  NOR is saved in dense form, column-first

3.1.6 Generation of files to be read by Fedem

When the model has been solved, the user can click the *Export Fedem Data* button in the ModRed toolbar, thus invoking the `onExportData()` method in `modRed.py`. The full mass matrix will then be partitioned into external and internal parts as described in Section 2.4.5.

For visualising the solved model in Fedem, a `.ftl` file of the model is created and saved in the specified Fedem directory. Binary `.fmx` files containg the reduced mass and stiffness matrix and gravity vectors are also generated.

3.2 Verifying Data From the ModRed Extension

In order to verify that all matrix handling is performed correctly, an IronPython Unit Test framework has been set up. Here, all methods that are involved in matrix calculations are tested for common errors for ensuring correct code and matrix calculations. An excerpt of what is tested is shown in the following section.

Verification of Matrices

The exported matrices from ANSYS in MatrixMarket Format need to be read by the ModRed extension in order to generate the needed Fedem files. As the used math library `Math.NET.Numerics.LinearAlgebra` [20] does not have any built-in functions for reading MatrixMarket formatted files, this method has to be written customly for the ModRed app. Thus, it is important that the matrices are read correctly before further calculations.

Testing that the matrix is read properly could be done through unit testing. The following code snippet shows how it is verified that the matrix is correctly read into IronPython by verifying the number of rows and columns on a test file, as well as checking some entries custom-picked from the ASCII-formatted `.mmf` file.

```
1  def test_readMMFMatrix_full(self):
2      path = self._resourcesFolder_ + "\\beam.nas" + "\\Ansys" + "\\M_full.mmf"
3      mat = ModRed.readMMFMatrix(path, "full")
4
5      # Checking if dimensions are correct
6      self.assertEqual(mat.RowCount, 630)
7      self.assertEqual(mat.ColumnCount, 630)
8
9      # Checking some values at the boundaries:
10     self.assertEqual(mat[0,0], 8.688888888888170E-02)
11     self.assertEqual(mat[0, 629], 0)
12     self.assertEqual(mat[629, 0], 0)
13     self.assertEqual(mat[629, 629], 1.448148148148030E-16)
14
15     # Testing random values:
16     self.assertEqual(mat[23,23], 1.448148148148030E-16)
17     self.assertEqual(mat[285,243], 3.620370370370130E-07)
18     self.assertEqual(mat[291,291], 5.792592592592110E-06)
```

Testing for Symmetry

The generated mass and stiffness matrix should be symmetrical, both for the full and for the reduced versions. This is verified by the following simple test, which is also performed on the reduced mass matrix:

```
1  def test_readMMFMatrix_full_isSymmetric(self):
2      path = self._resourcesFolder_ + "\\beam.nas" + "\\Ansys" + "\\M_full.mmf"
3      mat = ModRed.readMMFMatrix(path, "full")
4
5      # Checking if matrix is symmetric:
6      for row in range(0, mat.RowCount):
7          for col in range(0, mat.ColumnCount):
8              self.assertEqual(mat[row, col], mat[col, row])
```

Testing for Consistent Mass

In order to verify that the mass matrix has a consistent mass after reduction, one can compute the total mass of the system by pre- and post-multiplying the system with unit translation in x, y and z direction:

$$m_{\text{total}} = \mathbf{u}_{\text{trans}}^T \mathbf{m} \mathbf{u}_{\text{trans}} \tag{3.2}$$

This is then compared with the mass of the model computed by Fedem. The same test is done with the full mass matrices by the following unit test:

```
1   def test_readMMFMatrix_red_correctMass( self ):
2       """
3       Applying unit translation in x, y, z direction
4       and verifying that it is the same mass as computed in Fedem
5       """
6       path = self._resourcesFolder_ + "\\beam.nas" + "\\Ansys" + "\\M_red.mmf"
7       mat = ModRed.readMMFMatrix(path, "sub")
8       correct_mass = 3.12800E+01
9
10      for i in [1, 2, 3]:
11          u = ModRed.createUnitVector(i, 4, 2)
12          m = u.Transpose().Multiply(mat).Multiply(u)[0,0]
13          self.assertAlmostEqual(correct_mass, m)
```

Note that the `assertAlmostEqual` method is used, as there will be a small difference in results because of floating point precision. The `assertAlmostEqual` method tests for equality down to 7 decimal places by default.

Testing if matrix is diagonal

As both mass and stiffness matrices will be diagonal, one could check if elements not on the diagonal are zero. This is best tested after the matrix has been partitioned, as the partitioned matrix should remain diagonal.

```
1   def test_massMatrix_diagonality ( self ):
2       #
3       # Verifying that a lumped matrix is diagonal
4       #
5       mat = ModRed.readMMFMatrix(self._resourcesFolder + "\\beam.nas" + "\\Ansys" +
    "\\M_full_lumped.mmf", "full")
6       for row in range(0, mat.RowCount):
7           for col in range(0, mat.ColumnCount):
8               if row == col:
```

```
9                    self . assertTrue (mat[row,  col ]  !=  0.0  )
10               else :
11                   self . assertTrue (mat[row,  col ]  == 0.0)
```

Testing if Matrices are Partitioned Correctly

Testing if the matrices are partitioned correctly may only be partially done. This is due to the fact that Fedem, which is used as a reference, does only export the full, unpartitioned mass matrix $\mathbf{M}$, the $\mathbf{M}_{ee}$ part of the partitioned matrix, and the reduced mass matrix $\mathbf{m}$. ANSYS does only export the full mass matrix $\mathbf{M}$ and the reduced mass matrix $\mathbf{m}$. This implies that it could only be verified that the $\mathbf{M}_{ee}$ part of the matrix is partitioned correctly. It should be noted that this only tells us that $\mathbf{M}_{ee}$ is partioned in an identical way as done by the Fedem reducer, not that it is identical to what ANSYS does, because ANSYS does not export the partitioned matrix. An example of testing the partitionMatrix function is shown in the following:

Test of partitionMatrix function

```
1   def  test_partitionMatrix_beam_Mee ( self ):
2           # Testing  with  M_full  from beam.nas (medium model)
3           mapping = ModRed.readMappingFileFedem(self._resourcesFolder + "\\beam.nas" +
        "\\Fedem" + "\\MEQN.res")
4           M_full  = ModRed.readFedemMatrix(self._resourcesFolder  + "\\beam.nas" +
        "\\Fedem" + "\\M_full.res")
5           eNodes = [1,  2,  3,  4]
6           iNodes = [ i  for  i  in  range(5,  106)]
7            M_full_partitioned   = ModRed.partitionMatrix(M_full ,  eNodes, iNodes,  mapping, 6)
8           # Using M_ee from Fedem as reference
9           M_ee = ModRed.readFedemVector(self._resourcesFolder  + "\\beam.nas" +
        "\\Fedem" + "\\M_ee.res")
10
11          for  row  in  range(0,  M_ee.RowCount):
12              for  col  in  range(0,  M_ee.ColumnCount):
13                  self . assertAlmostEqual(  M_full_partitioned  [row,  col ],  M_ee[row, col ])
```

Chapter 4

Results

Testing has been performed with the three FE models listed in Table 4.1. All models are NASTRAN `.nas` files created in an external CAD system. For acquiring matrices from ANSYS, the model is imported through the *External Module* component and opened in ANSYS Mechanical. The ModRed extension is then used for selecting external nodes and exporting the needed matrices. To acquire matrices from Fedem, the same external nodes as defined in ANSYS have been selected, and the `fedem_reducer` module is run. The matrices are then read from the result file `fedem_reducer.res` for analyses.

	Small	Medium	Large
Fedem element	QUAD4	QUAD4	QUAD4
ANSYS element	SHELL181	SHELL181	SHELL181
Number of nodes	6	105	7869
Number of elements	2	80	7680
Material	Steel	Steel	Aluminium
Thickness [m]	0.02	0.02	0.01
Number of selected external nodes	2	4	4
File name	twoQUAD4	beam	plate

Table 4.1: FE models used for testing the ModRed extension

The small model can be seen in Figure 4.1. It is very simple, containing two 4-node shell elements, with node 1 and 3 chosen to be external. The model is used because it facilitates easier debugging of its corresponding matrices and degree of freedom ordering.

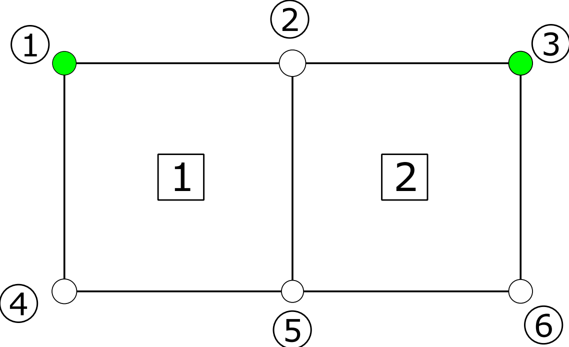

Figure 4.1: The small test model put togeter of two 4-node shell elements. The external nodes are marked green.

4.1 Similarity of Mass Matrices between ANSYS and Fedem

When analyses of dynamics is performed in Fedem, it is vital that the structural matrices for the models yield identical results, independent of what the origin of matrices are. When the ModRed app is to be used for enabling analyses in Fedem, FE models and their associated structural matrices are likely to be of two different origins:

1. The FE model will be made in an external CAD software and exported to a file format that could be imported by Fedem. The external nodes (triads) will then be selected in Fedem, and the Fedem reducer will be used to generate the reduced mass matrix $\mathbf{m}$ used in calculations.
2. The FE model is created or imported to ANSYS. The external nodes will then be selected in ANSYS Mechanical, and the internal ANSYS APDL solver will be used for calculating the reduced mass and stiffness matrix. These matrices will then be saved in a binary `.fmx` format and read by Fedem. The Fedem reducer will not be used.

In order to test the similarity of structural matrices in ANSYS and Fedem, structural matrices for the models seen in Table 4.1 have been generated in both ANSYS and Fedem and then compared. In order to quantify their internal difference, the Euclidean distance d between the sum of the two matrices is calculated. Then the relative difference α of the distance compared to the reference matrix is calculated:

$$d = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}(\mathbf{a}_{ij} - \mathbf{b}_{ij})^2}$$

$$\alpha = \frac{d}{\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbf{b}_{ij}}$$

Here, $\mathbf{a}_{ij}$ is the elements of the matrix that will be compared, and $\mathbf{b}_{ij}$ is the matrix used as reference. n is the total number of rows in the matrices. When using this method for comparing matrices, $\alpha = 0$ means the two matrices are identical. $\alpha = 1$ would imply there is a 100% difference between the matrices. I.e. a matrix filled with 2's compared to a matrix filled with 1's used as a reference, would give $\alpha = 1$. In the analysis, Fedem mass matrices are used as a reference in all models.

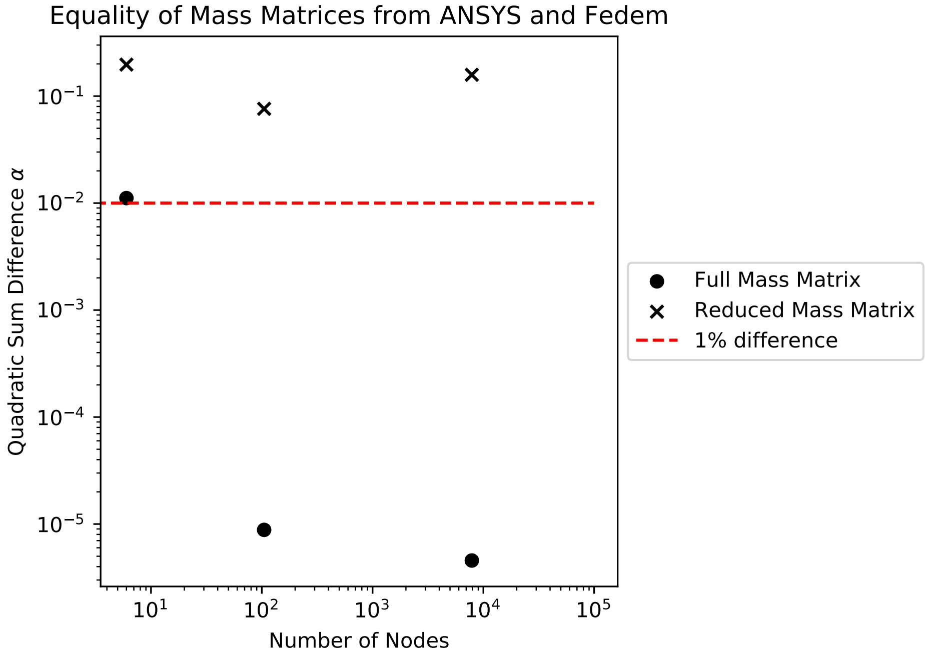

Figure 4.2: Percentage difference of full and reduced mass matrices from Fedem and ANSYS in relation to the number of nodes in the model

Figure 4.2 shows the percentage difference α of the full and reduced mass matrices of the test models. The full mass matrices yield good results for models with > 100 nodes, with $\alpha < 10^{-5}$. For the small model with only six nodes, about 1% difference is obtained. The reduced mass matrices however, show overall relatively poor results, with α values indicating $10 - 15\%$ difference between the reduced ANSYS and Fedem matrices. This means that if one were to set an acceptance limit of 1% difference, indicated in the figure, none of the reduced matrices would pass. Despite their poor α results, the reduced matrices are used in the following results, where they are, surprisingly, proven to yield some very good results. The results are further discussed in Section 5.1.

4.2 Testing for Consistent Mass in Models

As the full mass matrix is reduced in order to be used by Fedem, a key test for verifying that the reduced mass matrices are reduced correctly is to calculate the total mass of the model from the reduced mass matrix, and compare it to the total mass calculated from the full model. The model's total mass could be found by pre- and post-multiplying the mass matrix with unit translation in the x, y and z direction, similar to what is done in Section 2.4.3:

$$m_{\text{total}} = \mathbf{U}_{x,\text{red}}^T \mathbf{m} \mathbf{U}_{x,\text{red}} \tag{4.1}$$

The calculated total mass from reduced mass matrices in ANSYS is then compared to the calculated total mass from the full mass matrix from Fedem for all models. The full Fedem mass matrix is used as a reference in all calculations. The reference weight is found as an average of resulting weight of the full Fedem model after x, y and z translation. Mass calculated from the reduced mass matrices is also found as an average of x, y and z translation.

	Ref. [kg]	Diff. [%]
Small	312.8	-2.24×10^{-12}
Medium	31.28	-1.14×10^{-10}
Large	5.184	8.22×10^{-10}

Table 4.2: Mass of test models calculated from the reduced mass matrix compared to mass calculated from the full mass matrix

Table 4.2 shows resulting percentage difference of mass calculated from the reduced mass matrix compared to mass calculated from the full mass matrix. For all models, there is close to no difference between the calculated masses. This indicates that the reduction process has been successful.

4.3 Calculation of Gravity Vectors

Gravity vectors have been calculated for all three test models and compared to the gravity vector given by Fedem, which is calculated from the full mass matrix. For quantifying the vectors' similarity to the Fedem gravity vector, the relative difference α based on the Euclidean distance (4.1) is used. However, because we are now comparing two vectors, namely $\mathbf{g}_x$ from Fedem and ANSYS, the Euclidean distance d is found by summing all entries in the 1-dimensional vector. The same is done in y and z direction. The gravity vector from Fedem is used as reference for all models.

Both the full and reduced mass matrix from ANSYS have been used for calculating gravity vectors, following the equation set described in Section 2.4.3.

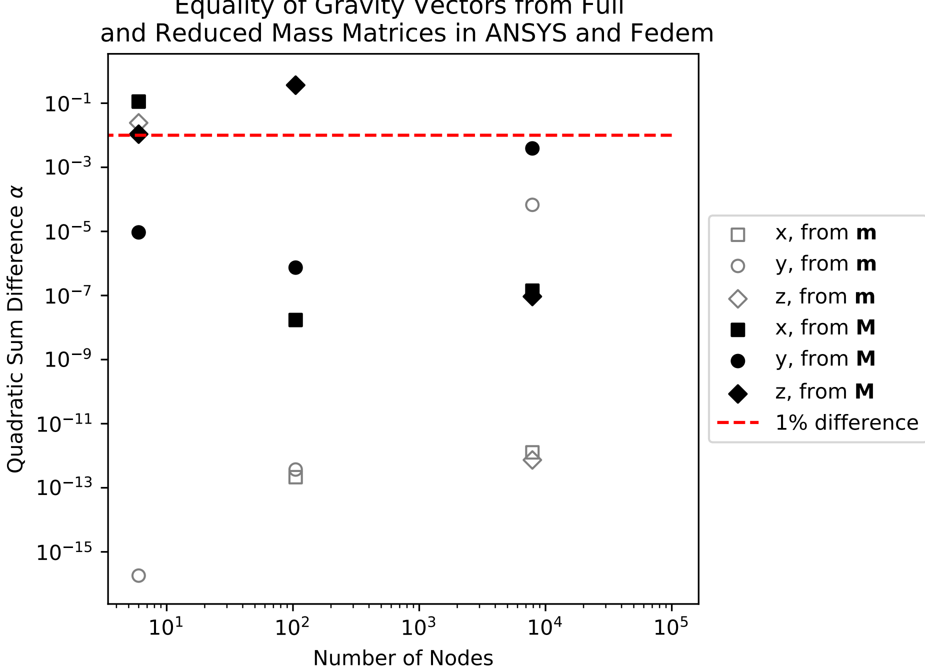

Figure 4.3: Relative difference of the gravity vector from Fedem and ANSYS, calculated with Fedem vectors used as reference

Figure 4.3 presents the resulting differences α between the gravity vectors. Resulting vectors when using the full mass matrix $\mathbf{M}$ are shown in filled black color, while resulting vectors when using the reduced mass matrix $\mathbf{m}$ are shown in un-filled grey. The results are very varied: some gravity vectors are nearly identical to gravity vectors from Fedem, namely the small model's x vector, the medium model's x and y vector and the large model's z and x vector, all from the reduced mass matrix. They have all $\alpha < 10^{-11}$. If one were to set $\alpha = 0.01$, corresponding to a 1% difference between ANSYS and Fedem vectors, as an acceptance limit, 12 out of 18 vectors are considered to have acceptable results. The worse result are obtained in the z vector from the medium model. Here, using both the full and reduced mass matrix gives 36.5% difference between ANSYS and Fedem vectors. Interestingly, using the reduced mass matrix generally gives better results than using the full mass matrix, with 5 vectors having $\alpha < 10^{-11}$.

There seem to be no ovious patterns regarding what direction gives the best results: both x, y and z vectors have α values close to the 1% difference mark, but are also almost identical to the Fedem reference in other models. Surprisingly, the number of nodes in the model seems to have almost no effect on the α values, although the large model gives acceptable results, with $\alpha < 3.9 \cdot 10^{-3}$ for all vectors.

4.4 Replicating Fedem Gravity Vectors

Due to the varied results when calculating gravity vectors from ANSYS matrices, gravity vectors using $\mathbf{M}$ and $\mathbf{H}$ matrix from Fedem have been calculated in order to replicate the results given by Fedem. The medium model is used for testing, and results are shown in Table 4.3. [1]

	$\mathbf{g}_x$	$\mathbf{g}_y$	$\mathbf{g}_z$
α	8.15×10^{-9}	8.94×10^{-9}	3.34×10^{-8}

Table 4.3: Verifying the calculations made in the ModRed extension by replicating gravity vectors from Fedem by using Fedem matrices in all calulations

The results are very close to what Fedem gives, with all vector directions having $\alpha < 3.34 \times 10^{-8}$. Even better results could probably be obtained if a higher precision on the exported Fedem matrices were chosen. Here, six decimals were used. These results verify that the calculations performed in the ModRed extension are able to replicate calculations performed by Fedem. Further comments to why the gravity vectors from using ANSYS matrices do not show better results are discussed in Section 5.1.

4.5 Generation of FTL Files in ANSYS

For models to be displayed in Fedem, a `.ftl` file of the model needs to be generated. It defines node coordinates as well as nodal and element connectivity in the model. When the *Export* button in the ModRed extension is pressed, a `.ftl` file of the model is generated along with binary `.fmx` files containing the reduced stiffness, mass and gravity vectors. For the simple twoQUAD4 model, the generated `.ftl` file is given as

FTL file for the twoQUAD4 model

```
1  FTLVERSION{4 ASCII}
2  #
3  # FTL file generated by ModRed at 2019-06-07 08:59:07
4  #
5
6  #
7  # Nodal coordinates
8  #
9  NODE{1 1 0.0 0.0 0.0}
10 NODE{2 0 1.0 0.0 0.0}
11 NODE{3 1 2.0 0.0 0.0}
12 NODE{4 0 0.0 -1.0 0.0}
```

[1]Performed in `test_results.py`, in method `test_quadraticSumDifference_gravityVector_beam_Fedem`

```
13 NODE{5 0 1.0 −1.0 0.0}
14 NODE{6 0 2.0 −1.0 0.0}
15 #
16 # Element definitions
17 #
18 QUAD4{1 5 4 1 2 {PTHICK 1} {PMAT 1}}
19 QUAD4{2 6 5 2 3 {PTHICK 1} {PMAT 1}}
20 #
21 # Local coordinate systems
22 #
23 PCOORDSYS{2 0 0 0 0 0 1 1 0 0}
24 #
25 # Material properties
26 #
27 PMAT{1 2.07e+11 8.02e+10 0.29 7820}
28 #
29 # Shell thicknesses
30 #
31 PTHICK{1 0.02}
32 #
33 # End of file
```

Currently, dummy values for the coordinate system, material properties and shell thickness is inserted, but should in the future be read from the ANSYS API. A future version of Fedem should also be able to detect that the model is reduced externally, and then use the `.fmx` files created by ModRed in analyses. This could for example be accomplished by reading the header of the `.ftl` file. If it says ”FTL file generated by ModRed at ...”, Fedem should recognize that it is an externally reduced model, and not start its own reducer.

Figure 4.4 shows the twoQUAD4 element opened in Fedem from the ModRed generated FTL file. The external nodes in the top left and right, with node ID 1 and 3, are correctly read and marked with a green triad.

4.6 Screencast Demonstration of the ModRed Extension

A video demonstrating the features of ModRed has been made and can be watched at `https://youtu.be/-E5c-ZojjsE`. It is demonstrated how to import models, the different settings for the extension, as well as how to import the generated .ftl file into Fedem.

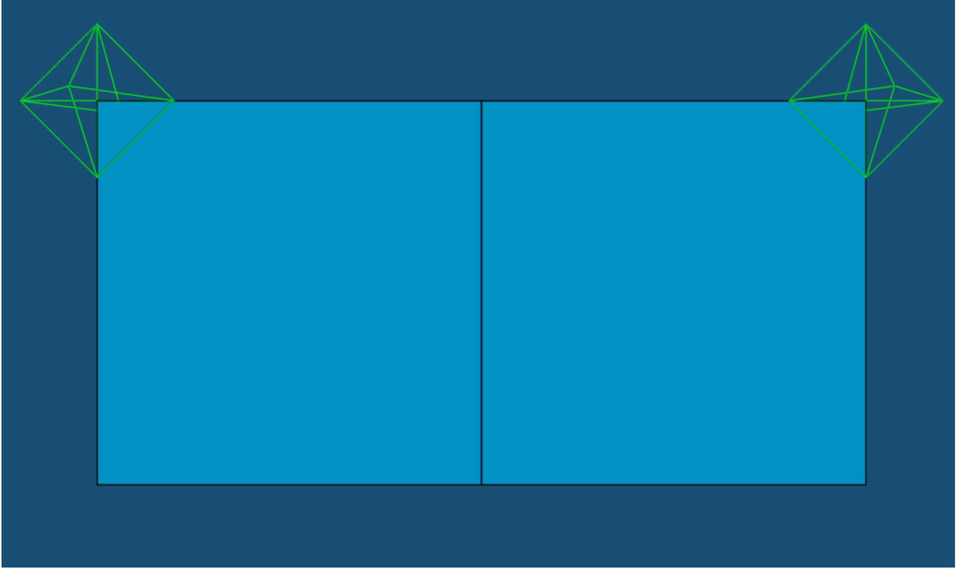

Figure 4.4: Visualisation of the twoQUAD4 element in Fedem, read from the ModRed generated FTL file

Chapter 5

Discussion and Future Work

5.1 Instability of Results

When observing the results obtained from the current version of the ModRed extension, there are some interesting observations to note:

- The full mass matrix $\mathbf{M}$ from ANSYS is shown to have much similiarity with the mass matrix generated by Fedem when the number of nodes is sufficiently high (over 10^2).
- The reduced mass matrices from ANSYS show less similarity, from 10 to 15% difference from Fedem matrices. However, when the same reduced mass matrices are used for calculating the gravity vector, some very good results are obtained - in fact, the gravity vectors from the reduced mass matrix are more similar to what Fedem gives than the gravity vectors calculated from the full mass matrix.
- Testing the mass matrices for mass consistency after partitioning shows us that no element data is lost in the process.
- If using $\mathbf{H}$ and $\mathbf{M}$ from Fedem when calculating gravity vectors, identical results are obtained from the ModRed extension as what is calculated by Fedem, indicating that the ModRed extension is replicating the calculations performed in Fedem.

From this, it seems likely that the reason for the inconsistent results comes from how ANSYS is structuring its mass matrices, and that this is not handled correctly by the ModRed extension. In the following, it is shown that partitioning of an ANSYS mass matrix, with the identical partitioning method as used by a Fedem mass matrix, does not generate an ANSYS mass matrix partitioned identically as the Fedem reference.

Test of Matrix Partitioning

```
1   def test_partitionMatrix_difference_twoQUAD4 ( self ):
2       """
```

```
3            Testing if Fedem and ANSYS matrices are partitioned correctly .
4            Using the partitioned Fedem matrix as reference , as this is proven to yield
5            the identical gravity vector as Fedem calculates .
6            """
7            dict_fedem = ModRed.readMappingFileFedem(self._resourcesFolder +
        "\\twoQUAD4" + "\\Fedem" + "\\MEQN.res")
8            dict_ansys = ModRed.readMappingFile(self. _resourcesFolder + "\\twoQUAD4" +
        "\\Ansys" + "\\M_full_lumped.mapping")
9            M_full_fedem = ModRed.readFedemMatrix(self._resourcesFolder + "\\twoQUAD4" +
        "\\Fedem" + "\\M_full.res")
10           M_full_ansys = ModRed.readMMFMatrix(self._resourcesFolder + "\\twoQUAD4" +
        "\\Ansys" + "\\M_full_lumped.mmf", "full")
11
12           eNodes = [1, 3]
13           iNodes = [2, 4, 5, 6]
14           fedem = ModRed.partitionMatrix(M_full_fedem, eNodes, iNodes, dict_fedem , 6,
        sorting ="equation")
15           ansys = ModRed.partitionMatrix( M_full_ansys , eNodes, iNodes, dict_ansys , 6,
        sorting ="equation")
16
17           for row in range(0, ansys.RowCount):
18               for col in range(0, ansys.ColumnCount):
19                   if row==col: # both matrices are diagonal
20                       self . assertAlmostEqual(fedem[row, col ], ansys[row, col ], places=1)
```

Result of Matrix Partitioning Test

```
r: 0,  c: 0,  fedem: 39.1,      ansys: 39.1
r: 1,  c: 1,  fedem: 39.1,      ansys: 39.1
r: 2,  c: 2,  fedem: 39.1,      ansys: 39.1
r: 3,  c: 3,  fedem: 0.278214,  ansys: 0.00130333333333
r: 4,  c: 4,  fedem: 0.393552,  ansys: 0.00130333333333
r: 5,  c: 5,  fedem: 0.651667,  ansys: 1.30333333333e−13
r: 6,  c: 6,  fedem: 39.1,      ansys: 39.1
r: 7,  c: 7,  fedem: 39.1,      ansys: 39.1
r: 8,  c: 8,  fedem: 39.1,      ansys: 39.1
r: 9,  c: 9,  fedem: 0.278214,  ansys: 0.00130333333333
r: 10, c: 10, fedem: 0.393552,  ansys: 0.00130333333333
r: 11, c: 11, fedem: 0.651667,  ansys: 1.30333333333e−13
r: 12, c: 12, fedem: 39.1,      ansys: 78.2
r: 13, c: 13, fedem: 39.1,      ansys: 78.2
r: 14, c: 14, fedem: 39.1,      ansys: 78.2
r: 15, c: 15, fedem: 0.278214,  ansys: 0.00260666666667
r: 16, c: 16, fedem: 0.393552,  ansys: 0.00260666666667
r: 17, c: 17, fedem: 0.651667,  ansys: 2.60666666667e−13
r: 18, c: 18, fedem: 39.1,      ansys: 39.1
r: 19, c: 19, fedem: 39.1,      ansys: 39.1
r: 20, c: 20, fedem: 39.1,      ansys: 39.1
r: 21, c: 21, fedem: 0.278214,  ansys: 0.00130333333333
```

```
r: 22,  c: 22,  fedem: 0.393552,     ansys: 0.00130333333333
r: 23,  c: 23,  fedem: 0.651667,     ansys: 1.30333333333e-13
r: 24,  c: 24,  fedem: 78.2,         ansys: 78.2
r: 25,  c: 25,  fedem: 78.2,         ansys: 78.2
r: 26,  c: 26,  fedem: 78.2,         ansys: 78.2
r: 27,  c: 27,  fedem: 0.556428,     ansys: 0.00260666666667
r: 28,  c: 28,  fedem: 0.787103,     ansys: 0.00260666666667
r: 29,  c: 29,  fedem: 1.30333,      ansys: 2.60666666667e-13
r: 30,  c: 30,  fedem: 78.2,         ansys: 39.1
r: 31,  c: 31,  fedem: 78.2,         ansys: 39.1
r: 32,  c: 32,  fedem: 78.2,         ansys: 39.1
r: 33,  c: 33,  fedem: 0.556428,     ansys: 0.00130333333333
r: 34,  c: 34,  fedem: 0.787103,     ansys: 0.00130333333333
r: 35,  c: 35,  fedem: 1.30333,      ansys: 1.30333333333e-13
```

The resulting printout of every diagonal item in the mass matrix shows that the partitioning of the two matrices is not identical. Using the partitioned Fedem matrix as a reference, one can see on line 13 that the partitioning of the ANSYS matrix can not be correct. The matrix entry should equal what Fedem gives (39.2), but is set to 78.2. This obviously wrong, as the translational DOFs from Fedem and ANSYS are similar (39.1 and 78.2) and should be set to the same position in both matrices. Attempts to resort the partitioned ANSYS matrix from other parameters than what Fedem uses - namely ascending equation number - has not yet proven successful.

The reason this issue is not straight-forward to resolve, is that it is limited what matrices could be exported from Fedem and ANSYS. From Fedem, one can obtain the full mass matrix, the $\mathbf{M}_{ee}$ part of the partitioned matrix and the reduced mass matrix. From ANSYS, you can only obtain the full and reduced mass matrices. This means that the only reference available for how to partition the mass matrix correctly, is $\mathbf{M}_{ee}$ from Fedem. It has therefore been assumed through the whole development process that ANSYS partitions the mass matrix identical to how Fedem does it. However, when it is proven that partitioning is handled differently in ANSYS, there is no $\mathbf{M}_{ee}$ matrix from ANSYS to use as reference, verifying that the partitioning has been done correctly.

This unresolved issue in matrix partitioning is believed to be the cause for the unstable results when calculating gravity vectors. The reason some of the gravity vectors gave very precise results is probably because large portions of the mass matrix *is* partitioned correctly, as seen in the example. The gravity vectors that gave very good results are therefore believed to have most parts of the mass matrix used for calculating them partitioned correctly.

The issue of matrix partitioning could also explain why the gravity vectors calculated from the reduced mass matrix give better results than using the full mass matrix for this. That is because when using the reduced mass matrix, the issue of matrix partitioning is removed, as the matrix is already partitioned internally by ANSYS before the reduced mass matrix is created from pre- and post-multiplying with $\mathbf{H}$

Due to the unstable results, it is difficult to make a general conclusion for the accuracy

of results obtained from ModRed. However, it is shown that the extension has the capability of producing precise results, with $\alpha < 10^{-11}$ for some of the gravity vectors. If the assumption that the root of unstable results lies in handling matrix partitioning, there should not be too much work in resolving the issue. In any way, more testing is needed, as well as some extra functionality discussed next, for the ModRed extension to be used effectively.

5.2 Future Work

In the following, suggestions for future development of the ModRed extension are listed. The list is based on experiences from developing the current version of the app, and will hopefully be a helpful reference in the future.

- Extending the elements supported by the ModRed extension should be a priority. Not only will it enable extra functionality for the extension, but will also extend the testing possibilities, thus making it easier to determine its accuracy compared to Fedem.
- When using ANSYS' automatic meshing capabilities, ANSYS is likely to create a mesh containing many different element types for the same model. Shell/membrane elements are automatically used for thin surfaces, different types of solid elements could be used for thicker surfaces, and contact elements are used where ANSYS finds it likely to be a contact point between two surfaces. If the model is to be exported to a `.ftl` file, the different elements created by ANSYS need to be mapped to a corresponding, similar Fedem element. This is currently not supported, as only SHELL181 elements are handled by the extension. ¨
- When importing the `.ftl` files into Fedem, it must be detected that the model has been externally reduced by ANSYS. This could for example be performed by reading the header of the FTL file, which will say *FTL file created by ModRed at [Date and time]*. If the text is detected, Fedem knows that the model is already reduced.

  In the current version of ModRed, the reduced mass matrix is used for calculating the gravity vectors needed by Fedem. Due to this, the calculations can be performed faster, as the **H** matrix is no longer needed for calculations, and thus does not have to be read line-for-line into the program. However, a future version of the ModRed extension should probably use the full mass matrix for gravity vector calculations as this should give better results.
- For future development, it is highly recommended to implement the code base in C# instead of the current IronPython language. The reasons for this are many:
  - Visual Studio, the text editor used for developing the ModRed extension, has a very limited support for the IronPython framework. For example is no auto-completion available, and method headers will normally not appear when calling methods written in the extension. This means that every input parameter

for methods used must be manually checked, resulting in tedious manual work, as well as code that is much more likely to contain errors.

- The ANSYS ACT API can be read by a C# environment, meaning that every method available through the API will be listed as auto-completion suggestions when handling ANSYS ACT objects. As the available ACT extension examples are relatively sparse, the development process often contains much trial-and-error for figuring out what methods from the API to use. Due to the lack of auto-completion in Visual Studio, all calls to the API must either be investigated through the ACT Console in ANSYS Mechanical, looked up online or in the Developer's Guide. This is very time-consuming, but the added help from the IDE if using C# would make this much easier.
- The Visual Studio debugger for IronPython is very unstable and is very likely to crash when connected to ANSYS Mechanical for debugging. Debugging the code is therefore often easier to perform with print statements.

- Overriding ANSYS' choice of elements for a mesh is proven to be a more complicated process than imagined. One can add the APDL command

  APDL command for overriding ANSYS' choice of mesh elements

```
et, matid, 181
```

  However, as this is a command sent directly to the APDL solver, the override of element type is not detected by ANSYS Mechanical, meaning there is no graphical feedback telling the user that the material type has been changed. In addition, the override of element type is not reflected in ANSYS ACT, meaning that internal node and element numbering could be altered at solve time.

- When performing model reduction through the ModRed extension, ANSYS Mechanical tells the user that *an unknown error occured*. However, the solver output from the APDL solver contains no errors, and all internal APDL solver files are generated. The error message is therefore probably occuring because of two reasons:

  1. When defining CMS as the solution type in APDL, we are accessing a solution type that is not supported by ANSYS Mechanical. Thus it is likely that the program defaults to show an error message, because it has no graphical interface for showing the CMS solution type.
  2. Many additional files are exported from ANSYS through APDL commands. Additionally, ANSYS moves all solver files to a temporary folder while the APDL solver is running. The export operations are therefore likely to be conflicting with ANSYS' internal file movement, which could be the cause for a default error message.

- In order to obtain 16-decimal precision on the exported mass and stiffness matrix, the full mass and the reduced stiffness and mass matrices are currently exported twice. This is because it is found that 16-decimal precision matrices are only exported in the Harwell-Boeing (`.hbmat`) format, accessed through the `fileaux2`

-> hbmat command. After export to the .hbmat format, the matrices are imported to an APDL Math matrix, and again exported to the Matrix Market Format. This format was found to be much easier to handle than the .hbmat format, because custom import methods for matrices had to be written, as the Math.NET package did not support matrix import in any of the formats. However, this double export is definitely a time- and space-consuming operation, and a method for reading .hbmat matrices should be made. Optionally, the APDL command *MWRITE should do exactly what we want: writing a matrix to a file in a user-formatted sequence. Unfortunately, it has not been possible to make this command work when writing commands directly to the ds.dat file.

Chapter 6

Conclusion

The ANSYS ACT extension ModRed has been developed in order to perform model reduction in ANSYS. By using it, one can access the reduced matrices as well as the needed transformation matrix after model reduction in ANSYS Mechanical. The matrices from ANSYS have been tested for equality against reference matrices from Fedem. The obtained results were varied:

- For large models with more than 100 nodes, the full mass matrices from Fedem and ANSYS are close to identical. However, the reduced mass matrices show overall poorer results, with the comparison method used indicating 10% to 15% difference between the reduced ANSYS and Fedem matrices.
- Despite being relatively unequal, the reduced matrices have identical mass to the full matrices, indicating that no information is lost in the reduction process. When calculating gravity vectors the reduced mass matrices surprisingly give overall better results than when using the full mass matrices in calculations.

A possible explanation to the varied results may be found in the way ANSYS structures their mass matrices differently than Fedem. It has been discovered that matrix partitioning in ANSYS is performed differently than what is the case in Fedem. This may explain why the results seem partially inconsistent. Extended testing using a wider range of test models, as well as resolving the issue on matrix partitioning should therefore be performed before one can conclude on the accuracy of results acquired from the extension. Despite the varied results, the extension has proven the potential to become an easy-to-use method for integrating ANSYS solver capabilities in the Fedem software.

References

[1] M. C. C. Bampton and R. R. Craig Jr. “Coupling of substructures for dynamic analyses.” In: *AIAA Journal* 6.7 (July 1968), pp. 1313–1319. ISSN: 0001-1452, 1533-385X. DOI: 10.2514/3.4741. URL: http://arc.aiaa.org/doi/10.2514/3.4741 (visited on 03/04/2019).

[2] Daniel J. Rixen. *Structural Dynamics*. München: Fachschaft Maschinenbau, Technische Universität München, Apr. 2017.

[3] ANSYS Inc. *ANSYS Mechanical APDL Theory Reference*. 2019. URL: https://ansyshelp.ansys.com/account/secured?returnurl=/Views/Secured/corp/v194/ans_thry/ans_thry.html (visited on 05/23/2019).

[4] ANSYS Inc. *Mechanical APDL Verification Manual*. 2019. URL: https://ansyshelp.ansys.com/account/secured?returnurl=/Views/Secured/corp/v194/ans_vm/Hlp_V_VMTOC.html (visited on 05/27/2019).

[5] ANSYS Inc. *Modal and Harmonic Frequency Analyses of an Automitive Suspension Assembly Using CMS: Results and Discussion*. 2019. URL: https://ansyshelp.ansys.com/account/secured?returnurl=/Views/Secured/corp/v194/ans_tec/tecautosuspresults.html (visited on 05/29/2019).

[6] IronPython. *IronPython.net*. IronPython. 2019. URL: https://ironpython.net/ (visited on 06/01/2019).

[7] ANSYS Inc. *ANSYS ACT Developer’s Guide*. 2019. URL: https://ansyshelp.ansys.com/account/secured?returnurl=/Views/Secured/corp/v194/act_dev/act_dev.html (visited on 05/29/2019).

[8] ANSYS Inc. *Mechanical APDL Theory Reference: Understanding Shape Function Labels*. 2019. URL: https://ansyshelp.ansys.com/account/secured?returnurl=/Views/Secured/corp/v194/ans_thry/understlabel.html (visited on 06/02/2019).

[9] ANSYS Inc. *ANSYS Mechanical APDL Element Reference*. 2019. URL: https://ansyshelp.ansys.com/account/secured?returnurl=/Views/Secured/corp/v194/ans_elem/Hlp_E_ElemTOC.html (visited on 05/30/2019).

[10] Fedem Technology. *Fedem Release 7.2 Theory Guide*. May 2019.

[11] Ole Ivar Sivertsen. *Virtual Testing of Mechanical Systems: Theories and Techniques*. Vol. 4. Advances in engineering. Lisse: Swets & Zeitlinger, 2001. ISBN: 90-265-1811-0.

[12] Fedem Technology. *Fedem Release R7.2 .1 User's Guide*. 2019.

[13] P. G. Bergan, M. K. Nygård, and R. O. Bjærum. "Free Formulation Elements with Drilling Freedoms for Stability Analysis of Shells". In: *Computational Mechanics of Nonlinear Response of Shells*. Ed. by Wilfried B. Krätzig and Eugenio Oñate. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 164–182. ISBN: 978-3-642-84045-6. DOI: 10.1007/978-3-642-84045-6_8. URL: https://doi.org/10.1007/978-3-642-84045-6_8.

[14] Pål G Bergan and Nygård, M. K. *A Quadrilateral Membrane Element with Rotational Freedoms*. Tokyo: Springer, 1986.

[15] Carlos A. Felippa and Carmelo Militello. "Membrane triangles with corner drilling freedoms—II. The ANDES element". In: *Finite Elements in Analysis and Design* 12.3 (Dec. 1992), pp. 189–201. ISSN: 0168874X. DOI: 10.1016/0168-874X(92)90034-A. URL: https://linkinghub.elsevier.com/retrieve/pii/0168874X9290034A (visited on 06/04/2019).

[16] B Skallerud and B Haugen. "Collapse of thin shell structures-stress resultant plasticity modelling within a co-rotated ANDES finite element formulation". In: *International Journal for Numerical Methods in Engineering* 46 (1999), p. 26.

[17] Ian Sommerville. *Software Engineering*. 10th ed., global ed. Boston Mass.: Pearson, 2016. ISBN: 978-1-292-09613-1.

[18] IronPython. *Unit testing framework — IronPython 2.7.2b1 documentation*. 2019. URL: https://ironpython-test.readthedocs.io/en/latest/library/unittest.html (visited on 06/01/2019).

[19] ANSYS Inc. *APDL Command Reference*. 2019. URL: https://ansyshelp.ansys.com/account/secured?returnurl=/Views/Secured/corp/v194/ans_cmd/Hlp_C_CmdTOC.html (visited on 05/30/2019).

[20] Math.NET. *MathNet.Numerics.LinearAlgebra - Math.NET Numerics Documentation*. 2019. URL: https://numerics.mathdotnet.com/api/MathNet.Numerics.LinearAlgebra/ (visited on 05/31/2019).

Appendix

A

Source Code for the ModRed ACT Extension

A.1 ModRed.py

ModRed.py

```
1  import clr
2  clr.AddReferenceToFileAndPath(r"C:\Program Files\ANSYS
       Inc\v192\Addins\ACT\bin\Win64\MathNet.Numerics.dll")    # Using the Math.NET dll.
       Can not use NumPy, as this is not supported by IronPython.
3  import MathNet.Numerics.LinearAlgebra as la     # math library
4  import os
5  import subprocess
6  import datetime           # For writing current date in generated FTL file
7  import ctypes
8  import cPickle as pickle         # For saving object to file
9  import System
10
11
12 def createUnitVector(pos, nnodes, cmodes=0):
13     '''
14     Creates a vector with unit length at the specified position pos.
15
16     :param pos: Int of position to give unit length. 1=UX, 2=UY, 3=UZ, 4=ROTX,
        5=ROTY, 6=ROTZ
17     :param nnodes: Int of number of nodes to create unit length for.
18     :param cmodes: Number of component nodes to add unit length
19     return matrix of size ((nnodes*6)+cmodes x 1)
20     '''
21     if (pos <= 0):
```

```
22            raise Exception("pos argument must be int > 0")
23
24        u = la.Matrix[System.Double].Build.Dense(nnodes*6 + cmodes, 1)      # initializing
          with zeroes
25        for i in range(pos-1, nnodes*6, 6):      # Adding 1 at position pos
26            u[i, 0] = 1
27        for i in range(nnodes*6, nnodes*6 + cmodes):      # Filling component mode
          entries with 0s
28            u[i, 0] = 0
29        return u
30
31
32
33
34    def onCreateModRed(analysis):
35        """
36        Adding the ModRed model object in the model tree
37
38        :param analysis: The currently active analysis
39        """
40        if str(analysis.AnalysisType) == "Modal":
41            # Use the analysis to create the APDL Based Result Evaluation result.
42            analysis.CreateLoadObject("modred", "ModRed")
43
44        else:
45            # Display an error message in the Mechanical message log.
46            ExtAPI.Application.LogError("Can only select ModRed for Modal analysis.")
47
48
49    def saveObject(obj, fname):
50        """
51        Saves the current object to _fedemDir_
52        """
53        testResourcesPath = ExtAPI.ExtensionManager.CurrentExtension.InstallDir +
          "\\test-resources"
54        with open(testResourcesPath + "\\" + fname + ".p", "wb") as input:
55            pickle.dump(obj, input, pickle.HIGHEST_PROTOCOL)
56
57
58    def onExportData(analysis):
59        '''
60        :param analysis: Ansys.ACT.Automation.Mechanical.Analysis object.
61        - Calculates gravity vectors from reduced mass matrix
62        - Creates FTL file for Fedem
63        - Creates FMX files from fmxWriter DLL
64        - All files are saved in solverData._fedemDir
65        '''
66
67        ExtAPI.Log.WriteMessage("Exporting data for Fedem...")
68        load = analysis.GetLoadObjects("ModRed")[0]
```

```
69      solverData = SolverData( analysis , load)              # Creating a SolverData object
         for the current analysis .
70
71      # Collecting exported matrices in Math.NET format
72      M_red = readMMFMatrix(analysis.WorkingDir + "M_red.mmf", "sub")
73      K_red = readMMFMatrix(analysis.WorkingDir + "K_red.mmf", "sub")
74
75      G = calculateGravityVectorReduced (M_red, len( solverData . _eNodes),
         int ( solverData . _cModes)) # Using reduced mass matrix
76      # Generating FTL file for Fedem
77      ftlFile = open(os.path . join ( solverData . _fedemDir, solverData . _filename + ". ftl ") ,
         "w")          # Creating a new ftl file
78      generateFTL(solverData . getNodes() , solverData . getElements () , solverData . _eNodes,
         ftlFile )        # Generating FTL file to be placed in the solver directory .
79      ExtAPI.Log.WriteMessage("FTL file " + str ( solverData . _filename ) + ". ftl written to
         " + solverData . _fedemDir)
80
81      # Generating FMX file for Fedem
82      generateFMX(mathNET2list(G), mathNET2list(M_red), mathNET2list(K_red), solverData )
83
84  def calculateGravityVectorReduced (M_red, nENodes, cmodes):
85      """
86      Calculates gravity vectors for Fedem from the reduced mass matrix .
87      :param M_red: The reduced mass matrix .
88      :param nENodes: Number of selected external nodes
89      """
90      # Creating reduced unit vectors
91      u_x = createUnitVector (1, nENodes, cmodes)
92      u_y = createUnitVector (2, nENodes, cmodes)
93      u_z = createUnitVector (3, nENodes, cmodes)
94      u = u_x . Append(u_y).Append(u_z)
95
96      G_red = M_red.Multiply(u)
97      return G_red
98
99
100
101
102
103 def calculateGravityVector (M_full, H, nENodes, nINodes, dofs ):
104     """
105     Calculates gravity vectors for Fedem from the full mass matrix and H matrix .
106     Follows equations from page 118 in " Virtual Testing of Mechanical Systems"
107     :param M_full: The full , partitioned mass matrix .
108     :param H: The H matrix
109     H =
110     [[ I 0],
111     [B, phi ]]
112     :param nENodes: Number of selected external nodes
113     :param nINodes: Number of selected internal nodes
```

```
114     :param dofs: Number of degrees of freedom for the element
115     """
116     # Extracting the partitions of the M matrix
117     partitions = extractPartitions (M_full, nENodes, dofs)
118     M_ee = partitions [0]
119     M_ei = partitions [1]
120     M_ie = partitions [2]
121     M_ii = partitions [3]
122
123     # Creating unit vectors:
124     u_x_e = createUnitVector (1, nENodes)
125     u_x_i = createUnitVector (1, nINodes)
126     u_y_e = createUnitVector (2, nENodes)
127     u_y_i = createUnitVector (2, nINodes)
128     u_z_e = createUnitVector (3, nENodes)
129     u_z_i = createUnitVector (3, nINodes)
130
131   # Gravitational forces:
132     G_x_i = M_ii.Multiply(u_x_i).Add(M_ie.Multiply(u_x_e))       # M_ii*u_x_i + M_ie *
          u_x_e
133     G_x_e = M_ei.Multiply(u_x_i).Add(M_ee.Multiply(u_x_e))
134     G_y_i = M_ii.Multiply(u_y_i).Add(M_ie.Multiply(u_y_e))
135     G_y_e = M_ei.Multiply(u_y_i).Add(M_ee.Multiply(u_y_e))
136     G_z_i = M_ii.Multiply(u_z_i).Add(M_ie.Multiply(u_z_e))
137     G_z_e = M_ei.Multiply(u_z_i).Add(M_ee.Multiply(u_z_e))
138
139     # Collecting in one matrix:
140     G_x = G_x_e.Stack(G_x_i)
141     G_y = G_y_e.Stack(G_y_i)
142     G_z = G_z_e.Stack(G_z_i)
143
144     # Reducing to unit gravitational acceleration on the full matrix:
145     G_x = H.Transpose().Multiply(G_x)
146     G_y = H.Transpose().Multiply(G_y)
147     G_z = H.Transpose().Multiply(G_z)
148
149     # Collecting gravity vectors in one matrix
150     G = G_x.Append(G_y).Append(G_z)
151     return G
152
153
154
155 def extractPartitions (mat, numENodes, dofs):
156     """
157     Extracts the partitions
158     M =
159     [[M_ee, M_ei],
160     [M_ie, M_ii]]
161     from the matrix mat
162     M_ee of size (numENodes*dofs, numENodes*dofs)
```

```
163
164         :param mat: Symmetric Math.NET matrix already partitioned into M_ee, M_ei, ...
165         :param eNodes: Number of selected external nodes
166         :param dofs: Number of degrees of freedom per node
167         """
168         nRows = mat.RowCount
169         nCols = mat.ColumnCount
170
171         M_ee = mat.SubMatrix(0, numENodes*dofs, 0, numENodes*dofs)     # Top left of M
172         M_ei = mat.SubMatrix(0, numENodes*dofs, numENodes*dofs, nCols - numENodes * dofs)
              # Top right
173         M_ie = mat.SubMatrix(numENodes*dofs, nRows - numENodes * dofs, 0,
             numENodes*dofs)    # Bottom left
174         M_ii = mat.SubMatrix(numENodes*dofs, nRows - numENodes * dofs, numENodes*dofs,
             nCols - numENodes*dofs)     # Bottom right
175         return [ M_ee, M_ei, M_ie, M_ii ]
176
177     def mathNET2list(matrix):
178         """
179         Returns a Python list of the items in the matrix
180
181         :param matrix: Math.NET matrix
182         return One-dimensional list of items in matrix saved column-first.
183         """
184         list = [0] * (matrix.ColumnCount * matrix.RowCount)
185         index = 0
186         for col in range(0, matrix.ColumnCount):
187             for row in range(0, matrix.RowCount):
188                 list[index] = matrix[row, col]
189                 index += 1
190         return list
191
192
193
194     def onSolve(load, stream):
195         """
196         Adds the required APDL commands to the solver input (ds.dat) file. Activates when
             the user hits "Solve"
197
198         :param load: the load associated to the callback. Interface IUserLoad
199         :param stream: a System.IO.StringWriter object, to which solver commands should be
             appended (represents the ds.dat file)
200         """
201
202         # Collecting user input:
203         nodeSelectionMethod = load.Properties["Generation"].Properties["Geometry"].Value
204         method = load.Properties["Method"].Properties["ReductionMethod"].Value
205         filename =
             load.Properties["Method"].Properties["ReductionMethod"].Properties["Name"].Value
             # string
```

```
206         nmode =
             load. Properties [”Method”]. Properties [”ReductionMethod”]. Properties [”Nmode”].Value
                 #  string
207         isUseLumped =
             load. Properties [”Method”]. Properties [”ReductionMethod”]. Properties [”LumpedMatrix”].Value
                 #  string
208         eNodes = []
209         if  (nodeSelectionMethod==”Geometry”):
210             eNodes = getENodes(load)        #  List [ int ]
211         if  (nodeSelectionMethod==”RBE2 Import”):
212             eNodes = getRBE2Nodes(load)
213         isExport  = load . Properties [”Export” ].  Properties [”IsExport” ]. Value
214         if  ( isExport  == ”Yes”):
215             isExport  = True
216         else :
217             isExport  = False
218
219         stiffnessName  = ”Kmat”
220         massName = ”Mmat”
221         matrixFormat = ”MMF”        # Set  this  to  ”. matrix”  for  lower  precision ,  but  more
             readable  matrices
222
223         #
224         # Writing  solver  commands to the  ds . dat  file
225         #
226         stream . WriteLine(” finish ”)
227         stream . WriteLine(”/ filname ,  ” + filename )       # Name of the  super  element  file  to
             be  generated
228         stream . WriteLine(”save”)
229         stream . WriteLine(”/ solu”)
230         stream . WriteLine(”antype ,  substr ”)              # Defining  a  substructure  analysis  type
             ( includes  both  Guyan and CMS)
231         if  (method == ”CMS”):  # If  CMSOPT needs to be set
232             cmsmeth = ”fix”
233             nmode = str(nmode)
234             freqb  = ””
235             freqe  = ””
236             fbddef  = ””
237             fbdval  = ””
238             iokey  = ”tcms”
239             if  ( iokey==”tcms”):
240                 stream . WriteLine(”outpr ,  nsol ,  all ”)            #  Print  command that must be
             defined  if  iokey==tcms
241             stream . WriteLine(”cmsopt,  ” + cmsmeth + ”,” + nmode + ”,” + freqb  + ”,” + freqe
             + ”,” + fbddef  + ”,” + fbdval  + ”,” + iokey )
242         sename = filename                  # name of  superelement  matrix  file
243         sematr  = str (2)               #  generate  stiffness  and mass matrices
244         sepr  = str (0)                 # Do not  print  superelement  matrices  or  load  vectors
245         sesst  = str (0)          # Do not  save  space  for  stress  stiffening
```

```
246         stream.WriteLine("seopt, " + sename + ", " + sematr + ", " + sepr + ", " + sesst +
            ", ,")
247         for nodeID in eNodes:
248             stream.WriteLine("m, " + str(nodeID) + ", all")
249         if (isUseLumped == "Yes"):
250             stream.WriteLine("lumpm, on")          # Using a lumped mass matrix
251         stream.WriteLine("alls")
252         stream.WriteLine("solve")
253         stream.WriteLine("save")
254         stream.WriteLine("finish")
255         stream.WriteLine("! Master node selection finished")
256
257         if (isExport):                   # Exporting matrices if selected
258             stream.WriteLine("! Exporting matrices ... ")
259             # Creating H matrix. H = [CST NOR]
260             stream.WriteLine("*dmat, cst, D, import, tcms, " + filename + ".tcms, cst")
              # Constraint mode data
261             stream.WriteLine("*dmat, nor, D, import, tcms, " + filename + ".tcms, nor")
              # Fixed-interface normal mode data
262             stream.WriteLine("save")
263             if (matrixFormat =="MMF"):
264                 stream.WriteLine("*export, cst, MMF, CST.mmf")          # Exporting
             constraint modes
265                 stream.WriteLine("*export, nor, MMF, NOR.mmf")          # Exporting
            fixed-interface normal mode data
266
267                 stream.WriteLine("*smat, nod2solv, D, import, full, " + filename + ".full,
            nod2solv")       # Importing the mapping vector internal -> solver ordering
268                 stream.WriteLine("*vec, mapback, I, import, full, " + filename + ".full,
            back")      # Importing the BACK nodal mapping vector for external -> internal
            ordering.
269
270                 stream.WriteLine("/aux2")          # Manipulating binary files
271                 stream.WriteLine("FILEAUX2, " + filename + ", sub")          # Specifying
            to dump reduced matrices file
272                 stream.WriteLine("HBMAT, M_red, hbmat, , ASCII, MASS, NO")          #
            Dumping reduced mass matrix to Harwell-Boeing format for 16-decimal precision
273                 stream.WriteLine("HBMAT, K_red, hbmat, , ASCII, STIFF, NO")     # Dumping
            reduced stiffness matrix
274
275                 stream.WriteLine("FILEAUX2, " + filename + ", full")          # Specifying to
            dump full mass matrix
276                 stream.WriteLine("HBMAT, M_full, hbmat, , ASCII, MASS, NO, YES")       #
            Dumping full mass matrix in Harwell-Boeing format
277                 stream.WriteLine("finish")
278                 stream.WriteLine("*smat, M_red, D, import, HBMAT, M_red.hbmat, ASCII")   #
            Creating sparse matrix from Harwell-Boeing format. Preserving 16-decimal precision
279                 stream.WriteLine("*smat, M_full, D, import, HBMAT, M_full.hbmat, ASCII")
280                 stream.WriteLine("*smat, K_red, D, import, HBMAT, K_red.hbmat, ASCII")
281
```

```
282                 stream.WriteLine("*export, M_red, MMF, M_red.mmf")           # Exporting
        to Matrix Market Format. Preserves 16-decimal precision
283                 stream.WriteLine("*export, M_full, MMF, M_full.mmf")
284                 stream.WriteLine("*export, K_red, MMF, K_red.mmf")
285
286             if (matrixFormat == ".matrix"):
287                 stream.WriteLine("*PRINT, Kmat, " + stiffnessName + ".matrix")           #
        Printing matrix directly in easy-to-read format
288                 stream.WriteLine("*PRINT, Mmat, " + massName + ".matrix")
289
290             stream.WriteLine("save")
291
292         stream.WriteLine("/eof")          # Quits correctly
293
294
295
296
297 def generateFMX(G, M_red, K_red, solverData):
298     """
299     This will save FMX files of G, M_red and K_red at the solver directory.
300
301     :param G: One-dimensional list of gravitational vectors saved column-first.
302     :param M_red: One-dimensional list of reduced mass matrix saved column-first.
303     :param K_red: One-dimensional list of reduced stiffness matrix saved column-first.
304     :param solverData: SolverData object.
305
306     """
307     # Converting to double slashes and in bytes representation
308     file = solverData._fedemDir + "\\" + solverData._filename
309     file = file.replace("\\", "\\\\")
310     file = bytes(file, 'utf-8')
311
312     # Skriver stivhetsmatrise
313     status = writeFMX(file, 1, K_red)
314
315     # Skriver massematrise
316     status = writeFMX(file, 2, M_red)
317
318     # Skriver gravitasjonskrefter
319     status = writeFMX(file, 3, G)
320
321     ExtAPI.Log.WriteMessage("FMX files written to " + solverData._fedemDir)
322
323
324
325 def writeFMX(file, ityp, data):
326     """
327     Writes a square matrix as a binary FMX-file for FEDEM, using the fmxWriter DLL
328
329     arg file : Full path to the fmx-file to be written
```

```
330        arg  ityp  :  1= stiffness  matrix,  2=mass matrix,  3=gravity  force  vectors
331        arg  data  :  Matrix  content,  column−wise storage
332        return  :  Zero on  success,  otherwise  negative
333        """
334        # Loading DLL for writing  FMX files:
335        extensionDir  = ExtAPI.ExtensionManager.CurrentExtension. InstallDir
336        dll  = ctypes. cdll .LoadLibrary( extensionDir  + "\\fmxWriter\\fmxWriter.dll")
337        cfil  =  file
338        ctyp  = ctypes. c_int ( ityp )
339        cdat  = ( ctypes . c_double*len( data ) ) ()
340        cdat [:]  = data
341        clen  = ctypes. c_int (len( data ) )
342        return  dll .WRITEFMX(cfil, ctypes.byref(ctyp),  cdat,  ctypes. byref( clen ),  len( file ) )
343
344
345  def  getENodes(load):
346        """
347        Returning  list  with  node numbers of  selected  external  nodes
348
349        :param load:  The current  load  object
350        : return  List [ int ]  Sorted  list  with  integers  representing  node numbers of  current
             selected  nodes.
351        """
352        selectedIDs  =
             load. Properties ["Generation" ].  Properties ["Geometry"]. Properties ["Geometry"].Value. Ids
353        return  sorted( selectedIDs )          #  List [ int ]
354
355
356
357  def  generateFTL(nodes,  elements,  eNodes,  file ):
358        '''
359        Generates  the  FTL  file  for  the  current  model to  be  used  by Fedem.
360        The FTL  file  is  placed  in  the  solverData . _fedemDir_
361
362        :param nodes:  list [Node] of  all  nodes  in  the  model.  Sorted  ascending  by  id
363        :param elements:  list [Element]  of  all  elements  in  the  model.  Sorted  ascending  by  id
364        :param eNodes:  list [ int ]  of  ids  of  external  nodes
365        :param  file :  FTL  file  object
366        '''
367        #
368        # Writing  file  header
369        #
370        date  = datetime . datetime .now(). strftime ("%Y−%m−%d %H:%M:%S")
371        file . write ("FTLVERSION{4 ASCII}\n")
372        string  ="#\n# FTL file  generated  by ModRed at " + str( date )  + "\n#\n\n"
373        file . write ( string )
374        string  = "#\n# Nodal  coordinates \n#"
375        file . write ( string  + "\n")
376
377        #
```

```
378     # Writing NODE part
379     #
380     for node in nodes:
381         nodeString = getNodeString(node, eNodes)
382         file . write (nodeString + "\n")
383
384     #
385     # Writing element definitions part
386     #
387     thicknessId = 1
388     materialId = 1
389     file . write ("#\n# Element definitions \n#\n")
390     for element in elements:
391         elString = getElementString (element, thicknessId , materialId )
392         file . write ( elString + "\n")
393     #
394     # Writing coordinate system part
395     #
396     file . write ("#\n# Local coordinate systems\n#\n")
397     file . write ("PCOORDSYS{2 0 0 0 0 0 1 1 0 0}" + "\n")
398
399     #
400     # Writing material properties part
401     #
402     file . write ("#\n# Material properties \n#\n")
403     file . write ("PMAT{1 2.07e+11 8.02e+10 0.29 7820}" + "\n")       # Dummy material
404
405     #
406     # Writing shell thicknesses part
407     #
408     file . write ("#\n# Shell thicknesses \n#\n")
409     file . write ("PTHICK{1 0.02}" + "\n")
410
411     #
412     # End of file
413     #
414     file . write ("#\n# End of file ")
415
416     file . close ()              # Closing the created FTL file
417
418
419 def getNodeString(node, eNodes):
420     """
421     Returns node string to write in FTL file .
422
423     :param id: id of node to write to string
424     :param nodes: sorted list [Node] of all nodes in the model. Sorted by node id
425     :param eNodes: list [ int ] of ids of selected external nodes
426     : return string of NODE part to be written in FTL file
427     """
```

```
428        string = "NODE{"
429        string += str(node._id)
430        string += " "
431        if(node._id in eNodes):
432            string += "1"
433        else:
434            string += "0"
435        string += " "
436        string += str(node._x) + " " + str(node._y) + " " + str(node._z)
437        string += "}"
438        return string
439
440
441    def getElementString(element, thicknessId, materialId):
442        """
443        Returns string of ELEMENT part to be written in FTL file
444
445        :param element: Element object of the element to write
446        :param thicknessID: id of thickness PTHICK
447        :param materialID: id of material PMAT
448        """
449        string = "QUAD4{"
450        string += str(element._id) + " "
451        string += str(element._nodes[0]._id) + " "
452        string += str(element._nodes[1]._id) + " "
453        string += str(element._nodes[2]._id) + " "
454        string += str(element._nodes[3]._id) + " "
455        string += "{PTHICK "
456        string += str(thicknessId) + "}"
457        string += " "
458        string += "{PMAT "
459        string += str(materialId) + "}"
460        string += "}"
461
462        return string
463
464
465
466
467    def isValidNoModes(entity, property):
468        '''
469        Returns true if valid input to component modes property. False if not.
470
471        :param entity: IUserObject. The entity containing the property.
472        :param property: ISimProperty. The property to check.
473
474        :return True if property is positive int. False if not.
475        '''
476        try:
477            #ExtAPI.Log.WriteMessage(str(int(property.Value) >= 0))
```

```
478             return int(property.Value) >= 0
479         except ValueError:
480             #ExtAPI.Log.WriteMessage(str(False))
481             return False
482 
483 def readMappingFileFedem(path):
484     """
485     Reads a Fedem mapping file that maps degree of freedom to equation.
486     The MEQN file is structured
487     column 0: dof       column 1: equation
488 
489     1   25
490     2   26
491     3   27   ...
492 
493     This means dof 1 equals equation 25.
494 
495     :param path: File path of MEQN.res file from Fedem
496     :return Dictionary on the form
497     dict = {
498     "dof": equation
499     }
500     """
501     file = open(path, "r")
502     line = file.readline()          # Reading header line
503     dict = {}
504     lines = file.readlines()
505     for line in lines:
506         line = line.split()
507         dof = int(line[0])
508         eq = int(line[1])
509         dict[dof] = eq
510     return dict
511 
512 def readMappingFile(path):
513     """
514     Reads an ANSYS .mapping file that maps equation to corresponding node.
515 
516     The .mapping file is structured:
517     column 0: equation      column 1: node      column 2: dof
518     Example:
519 
520     1   5   UX
521     2   5   UY
522     3   5   UZ
523 
524     This means equation 1 is related to node 5, dof UX
525     Relation node -> dof is
526     node 1, UX      => dof 1
527     node 1, UY      => dof 2
```

```
528     node 1, UZ      => dof 3
529     node 2, ROTX    => dof 4 ...
530
531     node 2, UX      => dof 7
532
533     :param path: File path of ".mapping" file
534     :param dofDict: Dictionary relating internal dof for a node to it corresponding
        numbering.
535     Example: "UX" = 1, "UY" = 2, "UZ" = 3, "ROTX" = 4
536     : return Dictionary
537     dict = {
538         "dof": "equation"
539     }
540     where "dof" is in ascending order from node numbers, i.e.:
541     dof = 1 => Node 1, degree of freedom nr 1 (UX)
542     dof = 2 => Node 1, degree of freedom nr 2 (UY)
543     dof = 7 => Node 2, degree of freedom nr 1 (UX)
544     dof = 12 => Node 2, degree of freedom nr 6 (ROTZ)
545     """
546     dofDict = {
547         "UX": 1,
548         "UY": 2,
549         "UZ": 3,
550         "ROTX": 4,
551         "ROTY": 5,
552         "ROTZ": 6
553     }
554     dict = {}
555     file = open(path, "r")
556     line = file.readline()      # Skipping first line
557     lines = file.readlines()
558     for line in lines:
559         line = line.split()
560         eq = int(line[0])
561         node = int(line[1])
562         dofText = line[2]
563         dof = node2dof(node, dofDict, 6, dofText)          # Currently hard-typing in
        6 dofs
564         dict[dof] = eq
565     return dict
566
567 def node2dof(node, dofDict, dofs, dofText):
568     """
569     Relates node number to dof number.
570
571     :param node: int of node number
572     :param dofs: Number of dofs for the node
573     :param dofText: string of dof in ANSYS .mapping file. Ex.: "UX", "UY",
574     """
575     dof = (node-1)*dofs + dofDict[dofText]
```

```
576         return dof
577
578
579
580
581
582     def nodes2Dofs(nodes, dofs):
583         """
584         # Relation dof -> node is
585         # dof = (node - 1) * dofs + a
586         # where a = [1, dofs]
587         #
588         # Example:
589         # nodes = [1, 5, 10] in elements with 6 dofs
590         # => returnDofs = [1, 2, 3, 4, 5, 6, 25, 26, 27, 28, 29, 30, 55, 56, 57, 58, 59, 60]
591         #
592         # Example 2:
593         # nodes = [1, 3]
594         # dofs = 2
595         # => returnDofs = [1, 2, 5, 6]
596         """
597         returnDofs = []
598         for node in nodes:
599             partDofs = range((node - 1) * dofs + 1, (node - 1) * dofs + (dofs + 1))
600             returnDofs += partDofs
601         return returnDofs
602
603
604     def dofs2Eqs(dofs, dict):
605         """
606         Returns a list with the corresponding equation for a node.
607
608         # Example:
609         # dofs = [1, 2, 3]
610         # dict = {
611             1: 2,
612             2: 3,
613             3: 1
614             }
615             returns [2, 3, 1]
616
617         :param dofs: List of dofs. 1-indexed. Length n
618         :param dict: Dictionary coupling dof to equation like
619             dict[dof] = eq
620         :param return: List of equations for the specified dofs. Length n. 1-indexed.
621         """
622         eqs = []
623         for dof in dofs:
624             eq = dict[dof]
625             eqs.append(eq)
```

```
626     return eqs
627
628
629
630
631 def partitionMatrix (mat, eNodes, iNodes, dict, dofs, sorting="equation"):
632     '''
633     Creates a partitioned matrix on the form
634     M =
635     [[Mee, Mei],
636     [Mii, Mii]]
637
638     :param mat: The Math.NET matrix to partition
639     :param eNodes: list [ int ] of the selected external nodes. Indexing starting at 1.
         Sorted.
640     :param iNodes: list [ int ] of internal nodes. 1-indexed. Sorted.
641     :param dict: Dictionary with the mapping of DOF to equation number in the matrix.
642         Example: dict [dof] = eq
643         NOTE: dof and equation in the dictionary is 1-indexed!
644     :param DOFs: Number of degrees of freedom per node.
645     :param sorting: How to sort the partitioned matrix.
646     sorting="equation" means the partitioned matrix is sorted after ascending equation
         numbers. This is what Fedem uses.
647     This is the default selection.
648     sorting="dof" means the partitioned matrix is sorted after ascending degree of
         freedom numbers.
649
650     :return partitioned matrix
651     '''
652
653     # Getting the dof numbering for the external (e) and internal (i) nodes
654     # Relation dof -> node is
655     # dof = (node - 1) * dofs + a
656     # where a = [1, dofs]
657     # Example:
658     # eNodes = [1, 5, 10] in elements with 6 dofs
659     # => eDofs = [1, 2, 3, 4, 5, 6, 31, 32, 33, 34, 35, 36, 61, 62, 63, 64, 65, 65]
660     eDofs = nodes2Dofs(eNodes, dofs)
661     iDofs = nodes2Dofs(iNodes, dofs)
662
663     # Mapping to solver ordering:
664     eEqs = dofs2Eqs(eDofs, dict)          # equations corresponding to external
         nodes. 1-indexed.
665     iEqs = dofs2Eqs(iDofs, dict)          # equations corresponding to internal
         nodes. 1-indexed.
666     if (sorting=="equation"):      #eEqs and iEqs are already sorted by dof number.
667         eEqs = sorted(eEqs)
668         iEqs = sorted(iEqs)
669
670     # Initializing partitioned matrix
```

```
671     nRows = (len(eNodes) + len(iNodes)) * dofs
672     nCols = nRows
673     matPart = la.Double.SparseMatrix(nRows, nCols) # Initializing with zeroes
674 
675     # Moving rows
676     row = 0
677     for e in eEqs:
678         matPart.SetRow(row, mat.Row(e-1))
679         row+=1
680     for i in iEqs:
681         matPart.SetRow(row, mat.Row(i-1))
682         row+=1
683     matCopy = la.Double.SparseMatrix(nRows, nCols)     # Initializing with zeroes
684     matPart.CopyTo(matCopy)   # Using a copy because values are accesed by reference .
685     # Moving columns
686     col = 0
687     for e in eEqs:
688         matPart.SetColumn(col, matCopy.Column(e-1))
689         col+=1
690     for i in iEqs:
691         matPart.SetColumn(col, matCopy.Column(i-1))
692         col+=1
693     return matPart
694 
695 
696 
697 
698 
699 def readFedemMatrix(filename):
700     """
701     Reading a Fedem matrix to MathNETmatrix for easier comparison with ANSYS matrices.
702     Written to import the full mass and stiffness matrix from Fedem
703 
704     :param filename: Path to matrix.
705     : return a sparse Math.NET matrix with contents of the matrix in path
706     """
707      file = open(filename, "r")       # Reading data
708      line = file . readline ()
709      line = line . split ()
710     nRows = int( line [0])               # Number of columns and rows are stored in the
        first line after the comments
711     nCols = int ( line [1])
712     mat = la.Double.SparseMatrix(nRows, nCols) # Initializing with zeros
713 
714     row = 0
715     col = 0
716     content = file . readlines ()
717     for line in range(0, len( content )):
718         splitted = content [ line ]. split ()
719         if (len( splitted ) != 0):
```

```
720                if (content[line].split()[0].startswith("+++")):
721                    row = int(content[line].split()[2]) - 1
722                    colLine = content[line + 1].split()
723                    valLine = content[line + 2].split()
724                    col = int(colLine[0]) - 1
725                    val = float(valLine[0])
726                    mat[row, col] = val
727        file.close()
728        return mat
729
730    def readFedemVector(path):
731        """
732        Reads vector from fedem_reducer.res file for comparison with vectors from ANSYS
733        Reads gravity vectors, as well as BMAT.res and PHI.res
734
735        :param path: Path to Fedem vector to read.
736        :return Sparse math.NET matrix of the Fedem vector in path
737        """
738        file = open(path, "r")
739        lines = file.readlines()
740        firstLine = lines[0].split()
741        nRows = int(firstLine[0])
742        nCols = int(firstLine[1])
743        mat = la.Double.SparseMatrix(nRows, nCols)    # Initializing with zeros
744
745        colIndexes = []
746        nextNumCols = 0
747        currNumCols = 0
748        row = 0
749        col = 0
750        startCol = 0
751        for i in range(0, len(lines)):
752            col = startCol
753            line = lines[i].split()
754            if (i != (len(lines) - 1)):              # if not at end of file
755                nextNumCols = len(lines[i + 1].split())
756            currNumCols = len(line)
757            if (currNumCols > nextNumCols):            # if new columns
758                startCol = int(lines[i + 1].split()[0]) - 1
759                continue
760            row = int(line[0]) - 1
761            vals = list(map(float, line))[1:]
762            for val in vals:
763                mat[row, col] = val
764                col += 1
765        file.close()
766        return mat
767
768
769    def readMMFMatrix(filename, format):
```

```
770        '''
771        Reading a dense or sparse Matrix Market file format into a Math.NET array.
772        NOTE:
773        − The matrices are saved column first !
774        − Matrices created from the .SUB binary ANSYS file are saved with the full (n x n)
            matrix.
775            This is the case for M_red and K_red files
776        − Matrices created from the .FULL binary ANSYS file have only the lower triangular
            part of the matrix saved.
777
778        :param filename: String of file name to the .mmf file. Dense .mmf matrices are
            saved column−first!
779        :param format: Format of the MMF matrix. "sub" for the reduced matrices. " full " for
            the full matrices.
780        : return Dense Math.NET matrix read from filename. Saved row−first.
781        '''
782        isFirstLine = False
783        nRows = 0
784        nCols = 0
785        file = open(filename, "r")      # Reading data
786        #
787        # Reading the lines defining dimensions of the matrix
788        #
789        dimLine = None
790        while ( isFirstLine == False):
791            line = file . readline ()
792            if (not line . startswith ('%')):
793                isFirstLine = True
794                line = line . split ()
795                nRows = int( line [0])          # Number of columns and rows are stored in
            the first line after the comments
796                nCols = int( line [1])
797                dimLine = line          # Saving for use in the next for loop
798
799        # Initializing Math.NET matrix
800        mat = la .Double.SparseMatrix(nRows, nCols)
801        isDenseFull = False
802        isSparseFull = False
803        isSparseSub = False
804
805        if (len(dimLine) == 2 and format=="full"):
806            isDenseFull = True
807        if (len(dimLine) == 3 and format=="full"):
808            isSparseFull = True
809        if (len(dimLine) == 3 and format=="sub"):
810            isSparseSub = True
811
812
813        # Filling Math.NET matrix with values from the file
814        row = 0
```

```
815        col = 0
816        for line in file :
817            if (isDenseFull):            # If we are reading a dense MMF matrix from ANSYS
           SUB or full file
818                val = float ( line )
819                mat[row, col] = val
820
821                if (row != nRows − 1 ):        # Updating counting variables
822                    row += 1
823                else :
824                    col += 1
825                    row = 0
826            if (isSparseSub or isSparseFull ):        # We are reading a sparse MMF matrix
           from ANSYS FULL file
827                line = line . split ()
828                i = int( line [0]) − 1                # Reading indexes . Is 1−indexed in the
           MMF file
829                j = int( line [1]) − 1
830                val = float ( line [2])
831                mat[i , j ] = val
832                mat[j , i ] = val                # sparse MMF matrix is saved in lower
            triangular form
833        file . close ()
834        return mat
835
836
837
838
839    def getRBE2Nodes(load):
840        '''
841        Imports RBE2 nodes from NASTRAN file and returns list node ids of RBE2 nodes.
842        "CompID" is the string from the Workbench site, when pressing on external model,
           under "General" tab. Typically "Setup X"
843        " Identifier " is the string under identifier tab when double−clicking the external
           model. Typically "File1 ".
844        :param load: The current load object
845
846        : return List [ int ] Sorted list with integer nodal ids of found RBE2 nodes in
           NASTRAN file.
847        '''
848        compID =
           load. Properties ["Generation" ]. Properties ["Geometry"]. Properties ["CompID"].Value
849        identifier =
           load . Properties ["Generation" ]. Properties ["Geometry"]. Properties [" Identifier " ]. Value
850        commands = ExtAPI.DataModel.Project.Model.GetFECommandsRepository(compID,
           identifier)
851        rbe2Comms = commands.GetCommandsByName("RBE2")
852        rbe2Count = rbe2Comms.Count
853        nodeList = []
854        for i in range (0, rbe2Count):
```

```
855            cmd = rbe2Comms[i+1]
856            nodeList.append(int(cmd.GetArgument[2]))          # Node ID is on argument index
        two. Convert to int.
857        return sorted(nodeList)
858
859
860    class Element():
861        """
862        Class used for storing info on elements in a mesh
863        """
864
865        def __init__(self, id, nodes):
866            self._nodes = nodes          # list [Node] connected to the element
867            self._id = id                # Element id
868
869
870        def getNodes(self):
871            return self._nodes
872
873        def getId(self):
874            return self._id
875
876
877
878
879    class Node:
880        """
881        Data storage class for one node from a mesh
882        """
883
884        def __init__(self, id, dofs, elements, x, y, z):
885            """
886            :param id: Int of nodal id
887            :param dofs: Int of number of dofs for the node
888            :param elements: List[int] of element ids to the elements connected to the node
889            :param x: x coordinate
890            :param y: y coordinate
891            :param z: z coordinate
892            """
893            self._id = id
894            self._dofs = dofs
895            self._elements = elements
896            self._x = x
897            self._y = y
898            self._z = z
899
900
901        def getId():
902            return self._id
903
```

```
904     def getDofs():
905         return self._dofs
906 
907     def getElements():
908         return self._elements
909 
910 
911 
912 class SolverData:
913     #
914     # Class used to store data from the solver and ModRed application.
915     #
916 
917     def __init__(self, analysis, load):
918         #
919         # Collecting user input
920         #
921         self._dofs = 6          # Currently hard-typing in number of dofs
922         self._filename =
          load.Properties["Method"].Properties["ReductionMethod"].Properties["Name"].Value
          # string
923         self._workingDir = analysis.WorkingDir
924         self._fedemDir =
          load.Properties["Export"].Properties["IsExport"].Properties["FileDir"].Value
925         self._testResourcesDir = ExtAPI.ExtensionManager.CurrentExtension.InstallDir +
          "\\test-resources"
926         self._eNodes = getENodes(load)          # List[int] of selected external nodes.
          Sorted
927         self._cModes =
          int(load.Properties["Method"].Properties["ReductionMethod"].Properties["Nmode"].Value)
            # Number of component modes
928 
929         self._elements = self.getElementsFromMesh(analysis.MeshData)
          #List[Element]
930         self._nodes = self.getNodesFromElements(self._elements)      # list[Node]
931 
932         self._nRows = len(self._nodes) * self._dofs        # Number of rows in the full
          mass matrix
933         self._nCols = self._nRows          # Quadratic matrix
934 
935 
936 
937     def saveObject(self, obj, fname):
938         """
939         Saves the current object to _fedemDir
940         """
941         with open(self._testResourcesDir + "\\" + fname + ".p", "wb") as file:
942             pickle.dump(obj, file, pickle.HIGHEST_PROTOCOL)
943 
944     def getNodes(self):
```

```
945             """
946             Return sort list [Node] of nodes in the model. Sorted ascending by id
947             """
948             return sorted( self . _nodes, key=lambda x: x._id )
949
950
951         def getElements( self ):
952             """
953             Return sorted list [Element] of elements in the model. Sorted ascending by id
954             """
955             return sorted( self . _elements , key=lambda x: x._id )
956
957
958         def getNodesFromElements(self, elements ):
959             """
960             Creates list [Node] of nodes connected to the element.
961
962             :param elements: List [Element]
963             : return List [Node] with all nodes in the model
964             """
965             nodes = []
966             for e in elements:
967                 for n in e.getNodes():
968                     if not any(x._id == n._id for x in nodes):
969                         nodes.append(n)
970             return nodes
971
972         def getNodeIds( self ):
973             """
974             : return Sorted list [ int ] of all node ids in the model
975             """
976             ids = []
977             for node in self . _nodes:
978                 ids .append(node._id )
979             return sorted( ids )
980
981
982         def getElementsFromMesh(self, mesh):
983             """
984             Returns list of custom Element objects containing all elements in the model.
985
986             :param mesh: ANSYS MeshData object
987             : return List [Element]
988             """
989             returnElements = []
990             ansysElements = mesh.Elements
991             for e in ansysElements:
992                 id = e.Id
993
994                 nodeList = []
```

```
995                for node in e.Nodes:
996                    nodeId = node.Id
997                    elements = node.ConnectedElementIds             #List[int] of ids of
         the connected elements
998                    node = Node(nodeId, 6, elements, node.X, node.Y, node.Z)            #
         Currently hard-typing in dofs per node. Should be changed to be read from ANSYS
         API
999                    nodeList.append(node)
1000
1001               element = Element(id, nodeList)
1002               returnElements.append(element)
1003           return returnElements
1004
1005
1006
1007   class Results:
1008       """
1009       Class for calculating results from the model reduction extension
1010       """
1011       def __init__(self):
1012           self._resourcesFolder = os.path.dirname(os.path.realpath(__file__)) +
        "\\test-resources\\"
1013
1014
1015
1016       def quadraticSumDifference(self, mat, refMat):
1017           """
1018           Quadratic sum of the difference of all elements in n x n matrix
1019
1020           :param mat: Matrix that will be compared
1021           :param refMat: Matrix used as reference for comparison
1022           :return percentage difference on quadratic sum of mat, compared to refMat
1023           """
1024           matSum = self.matrixSum(mat)
1025           refMatSum = self.matrixSum(refMat)
1026           squared = (matSum - refMatSum) * (matSum - refMatSum)
1027           diff = math.sqrt(squared)
1028           percentage = diff/refMatSum
1029           return percentage
1030
1031
1032       def quadraticSumDifferenceOfVector(self, vec, refVec):
1033           """
1034           Quadratic sum of the difference of two vectors of size (n x 1)
1035
1036           :param : Vector that will be compared
1037           :param refVec: Vector used as reference for comparison
1038           :return percentage difference on quadratic sum of vec, compared to refVec
1039           """
1040           vecSum = vec.Sum()
```

```
1041            refVecSum = refVec.Sum()
1042            squared = (vecSum − refVecSum) ∗ (vecSum − refVecSum)
1043            diff = math.sqrt(squared)
1044            percentage = diff /refVecSum
1045            return percentage
1046
1047
1048        def matrixSum(self, mat):
1049            """
1050            Calculating the matrix sum of a mat.NET matrix
1051            :param mat: The matrix to calculate the sum of
1052            :return float Sum of all elements in the matrix
1053            """
1054            rowSums = mat.RowSums()
1055            return sum(rowSums)
1056
1057
1058        def calculateMassFromMassMatrix(self, mat, reduced=True, cmodes=2):
1059            """
1060            Calculates the mass of a mass matrix mat by applying unit translation :
1061            m_tot = u^T ∗ mat ∗ u
1062
1063            :param mat: The mass matrix
1064            :param reduced: Indicating if it is a reduced mass matrix. Defaults to True.
1065            :param cmodes: Number of component modes in the reduced mass matrix. Defaults
        to 2
1066            """
1067            if (reduced==True):
1068                nRows = mat.RowCount
1069                nNodes = int((nRows − cmodes) / 6)          # Currently assuming 6 DOFs per
        node
1070                u_x = ModRed.createUnitVector(1, nNodes, cmodes)
1071                u_y = ModRed.createUnitVector(2, nNodes, cmodes)
1072                u_z = ModRed.createUnitVector(3, nNodes, cmodes)
1073                mass_x = u_x.Transpose().Multiply(mat).Multiply(u_x) [0,0]
1074                mass_y = u_y.Transpose().Multiply(mat).Multiply(u_y) [0,0]
1075                mass_z = u_z.Transpose().Multiply(mat).Multiply(u_z) [0,0]
1076                return [mass_x, mass_y, mass_z]
1077            else:
1078                nRows = mat.RowCount
1079                nNodes = int(nRows/6)       # currently assuming 6 dofs per node
1080                u_x = ModRed.createUnitVector(1, nNodes)
1081                u_y = ModRed.createUnitVector(2, nNodes)
1082                u_z = ModRed.createUnitVector(3, nNodes)
1083                mass_x = u_x.Transpose().Multiply(mat).Multiply(u_x) [0,0]
1084                mass_y = u_y.Transpose().Multiply(mat).Multiply(u_y) [0,0]
1085                mass_z = u_z.Transpose().Multiply(mat).Multiply(u_z) [0,0]
1086                return [mass_x, mass_y, mass_z]
```

A.2 ModRed.xml

ModRed.xml

```
1  <extension version="1" name="ModRed">
2    <!-- This is the GUID for the ModRed extension. It must remain the same independently
     ↪ of
3    the extension version or name. It is used to uniquely identify this extension. The
     ↪ shortId
4    attribute is needed for compatibility with old projects, it is the name of the
     ↪ extension
5    before adding this GUID. -->
6    <guid shortid="modred">D0B4EDE5-4151-4EC1-96B4-BDA4410E0CDE</guid>
7          <script src="ModRed.py" compiled="true"/>
8          <interface context="Mechanical">
9                  <images>images</images>
10                 <toolbar name="modred" caption="Fedem Reduction">
11                         <entry name="Fedem Reduction" icon="fedem-transp" >
12         <callbacks>
13           <onclick>onCreateModRed</onclick>
14         </callbacks>
15                         </entry>
16       <separator />
17       <entry name="Export" caption="Generate Fedem Data" icon="fedem-export" >
18         <callbacks>
19           <onclick>onExportData</onclick>
20         </callbacks>
21       </entry >
22                 </toolbar>
23         </interface>
24
25         <simdata context="Mechanical">
26                 <!-- defining the object that is inserted under "Model" in the project
     ↪ tree -->
27                 <load name="modred" version="1" caption="Fedem Reduction"
     ↪ icon="fedem-transp"
28                         isload="true" color="#0000FF" contextual="true">
29                         <callbacks>
30         <!-- Allowing the user to right-click Fedem icon in the tree and select "Export
     ↪ Fedem Data" -->
31         <action name="onExportData" caption="Export Fedem Data"
     ↪ icon="fedem-export">onExportData</action>
32                                 <getsolvecommands>onSolve</getsolvecommands><!--
     ↪ Running the specified function when the user hits "solve"-->
33                         </callbacks>
34
35                         <propertygroup name="Generation"
36                                 caption="Selection of Master Nodes"
37                                 display="caption">
```

```
38                              <propertygroup name="Geometry" caption="Pick Master
    ↪ Nodes By..."  control="select"  display="Property"  default="Geometry">
39                                      < attributes  options="Geometry,RBE2 Import" />
40                                      <property name="Geometry" control="scoping"
41                                              caption="Geometry Selection"
    ↪ visibleon="Geometry">
42                                              < attributes   selection_filter  ="node" />
43                                      </property>
44                                      <property name="CompID" control="String"
    ↪ caption="Component ID" visibleon="RBE2 Import" />
45                                      <property name=" Identifier "  control="String"
    ↪ caption=" Identifier "  visibleon="RBE2 Import" />
46                              </propertygroup>
47                      </propertygroup>
48
49                      <propertygroup name="Method" caption="Model Reduction Method"
50                              display="Caption">
51                              <propertygroup name="ReductionMethod"
    ↪ caption="Reduction Method" control="select"  display="Property"  default="CMS">
52                                      < attributes  options="Guyan,CMS" />
53                                      <property name="Nmode" caption="Num. of
    ↪ Component Modes to Extract"
54                                              control="int"  default="2"
    ↪ visibleon="CMS">
55                                              <callbacks>
56
    ↪ <isvalid>isValidNoModes</isvalid>
57                                              </callbacks>
58                                      </property>
59                                      <property name="Name" caption="File Name"
    ↪ control="string"
60                                              default="master"
    ↪ visibleon="Guyan|CMS" />
61          <property name="LumpedMatrix" caption="Use lumped matrices?" control="select"
    ↪ default="No" >
62              < attributes  options="Yes,No" />
63          </property>
64                              </propertygroup>
65                      </propertygroup>
66
67                      <propertygroup name="Export"
68                              caption="Export Options"
69                              display="Caption">
70                              <propertygroup name="IsExport"  control="select"
    ↪ caption="Export Matrices  to  Fedem?" display="Property"
71                              default="Yes" >
72                                      < attributes  options="No,Yes" />
73                                      <property name="FileDir"  caption="Fedem File
    ↪ Directory"
74                                              control="folderopen"
```

```
75                                                   default=""
76                                                   visibleon ="Yes">
77                                          </property>
78                                 </propertygroup>
79                        </propertygroup>
80               </load>
81      </simdata>
82 </extension>
```

A.3 test_ModRed.py

test_ModRed.py

```
1
2
3  #
4  # Unit  tests  for  ModRed
5  #
6
7  import  clr
8  clr . AddReferenceToFileAndPath(r"C:\Program Files\ANSYS
       Inc\v192\Addins\ACT\bin\Win64\MathNet.Numerics.dll")   # Using the Math.NET dll.
       Can not use NumPy, as this is  not  supported  by  IronPython.
9  import  MathNet.Numerics.LinearAlgebra as  la
10 import  System
11 import  unittest
12 import  os
13 import  pickle
14 from System import  Array  as  sys_array
15
16 import  sys
17 sys . path . insert (0,
       r'C:\Users\adrian\Documents\Dokumenter\NTNU\Master\masters\ModRed\ModRed')
18 import  ModRed
19 from ModRed import Node
20 from ModRed import Element
21 from ModRed import SolverData
22
23
24
25 class  TestModRed(unittest . TestCase):
26
27     def  setUp( self ):
28         self . _resourcesFolder  = os . path . dirname(os . path . realpath (  __file__  )) +
       "\\ test -resources\\"
29
30         #
```

```
31            # Defining the twoQUAD4 model
32            #
33            self._node1 = Node(1, 6, [1], 0.0, 0.0, 0.0)
34            self._node2 = Node(2, 6, [1, 2], 1.0, 0.0, 0.0)
35            self._node3 = Node(3, 6, [2], 2.0, 0.0, 0.0)
36            self._node4 = Node(4, 6, [1], 0.0, −1.0, 0.0)
37            self._node5 = Node(5, 6, [1, 2], 1.0, −1.0, 0.0)
38            self._node6 = Node(6, 6, [2], 2.0, −1.0, 0.0)
39
40            self._element1 = Element(1, [self._node1, self._node2, self._node5,
          self._node4])     # clockwise
41            self._element2 = Element(2, [self._node2, self._node3, self._node6,
          self._node5])
42
43            self._elements = [self._element1, self._element2]
44            self._nodes = [self._node1, self._node2, self._node3, self._node4, self._node5,
          self._node6]
45            self._unsortedNodes = [self._node2, self._node1, self._node3, self._node6,
          self._node5, self._node4]
46
47
48
49
50
51            def array(*x): return sys_array[float](x)  # Helper function to create custom
          Math.NET matrices
52            self._mat_ = la.Double.Matrix.Build.DenseOfRowArrays(
53                array(1, 2, 3, 4, 5, 6, 7, 8, 9),
54                array(10, 11, 12, 13, 14, 15, 16, 17, 18),
55                array(19, 20, 21, 22, 23, 24, 25, 26, 27),
56                array(28, 29, 30, 31, 32, 33, 34, 35, 36),
57                array(37, 38, 39, 40, 41, 42, 43, 44, 45),
58                array(46, 47, 48, 49, 50, 51, 52, 53, 54),
59                array(55, 56, 57, 58, 59, 60, 61, 62, 63),
60                array(64, 65, 66, 67, 68, 69, 70, 71, 72),
61                array(73, 74, 75, 76, 77, 78, 79, 80, 81)
62            )
63            self._indexMat_ = la.Double.Matrix.Build.DenseOfRowArrays(
64                array(11, 12, 13, 14, 15, 16, 17, 18, 19),
65                array(21, 22, 23, 24, 25, 26, 27, 28, 29),
66                array(31, 32, 33, 34, 35, 36, 37, 38, 39),
67                array(41, 42, 43, 44, 45, 46, 47, 48, 49),
68                array(51, 52, 53, 54, 55, 56, 57, 58, 59),
69                array(61, 62, 63, 64, 65, 66, 67, 68, 69),
70                array(71, 72, 73, 74, 75, 76, 77, 78, 79),
71                array(81, 82, 83, 84, 85, 86, 87, 88, 89),
72                array(91, 92, 93, 94, 95, 96, 97, 98, 99)
73            )
74
75            self._indexMatSmall_ = la.Double.Matrix.Build.DenseOfRowArrays(
```

```
76                 array (11,  12,  13),
77                 array (21,  22,  23),
78                 array (31,  32,  33)
79                 )
80
81             self . _indexMat5x5_ = la . Double.Matrix . Build . DenseOfRowArrays(
82                 array (11,  12,  13,  14,  15),
83                 array (21,  22,  23,  24,  25),
84                 array (31,  32,  33,  34,  35),
85                 array (41,  42,  43,  44,  45),
86                 array (51,  52,  53,  54,  55)
87             )
88
89         def  tearDown(self ):
90             if  os . path . exists ( self . _resourcesFolder  + "\\testFTL. ftl "):
91                 os . remove( self . _resourcesFolder  + "\\testFTL. ftl ")
92
93
94
95         def  saveObject( self ,  obj ,  fname):
96             """
97             Saves  the  current  object  to  _fedemDir
98             """
99             with  open( self . _resourcesFolder  + "\\" + fname + ". p",  "wb") as  input:
100                pickle . dump(obj,  input ,  pickle . HIGHEST_PROTOCOL)
101
102
103
104
105        def  test_generateFTL ( self ):
106            file  = open( self . _resourcesFolder  + "\\testFTL. ftl ",  "w")
107            eNodes = [1,  3]
108            ModRed.generateFTL(self._nodes,  self . _elements ,  eNodes,  file )
109
110            # Need to  open  the  closed  file  again
111            with  open( self . _resourcesFolder  + "\\testFTL. ftl ",  "r")  as  file:
112                fileLines  =  file . readlines ()
113                file . close ()
114            with  open( self . _resourcesFolder  + "\\demoFTL.ftl", "r")  as  file:
115                correctFileLines  =  file . readlines ()
116                file . close ()
117            print (  fileLines  )
118            print (  correctFileLines  )
119            self . assertEqual ( len(  fileLines  ),  len(  correctFileLines  ))       # Should be  the
       same length
120            self . assertEqual (  fileLines [3 : ],   correctFileLines [3 : ])         # Content  is  equal
       (except  for  the  "generated  at  ... "  part
121
122
123
```

```
124     def  test_getNodeString ( self ):
125         node =  self . _node1
126         eNodes = [1,  3]
127
128          string  = ModRed.getNodeString(node, eNodes)
129          correctString  = "NODE{1 1 0.0 0.0 0.0}"
130          self . assertEqual ( string ,   correctString  )
131
132
133
134     def  test_getNodeString_notENode ( self ):
135         """
136          If  the  node id  is  not  in  eNodes,  it  should  be  marked with  0
137         """
138         node =  self . _node2
139         eNodes = [1,  3]
140
141          string  = ModRed.getNodeString(node, eNodes)
142          correctString  = "NODE{2 0 1.0 0.0 0.0}"
143          self . assertEqual ( string ,   correctString  )
144
145     def  test_getElementString  ( self ):
146         element  =  self . _element1
147          thicknessId  = 1
148         matId = 1
149          string  = ModRed.getElementString(element,  thicknessId ,  matId)
150          correctString  = "QUAD4{1 1 2 5 4 {PTHICK 1} {PMAT 1}}"
151
152          self . assertEqual ( string ,   correctString  )
153
154     def  test_getElementString_element2  ( self ):
155         element  =  self . _element2
156          thicknessId  = 1
157         matId = 1
158          string  = ModRed.getElementString(element,  thicknessId ,  matId)
159          correctString  = "QUAD4{2 2 3 6 5 {PTHICK 1} {PMAT 1}}"
160
161          self . assertEqual ( string ,   correctString  )
162
163
164
165     def  test_createUnitVector  ( self ):
166         """
167         Should  return  matrix  of  size  ((nnodes*6)+cmodes x 1)
168         """
169
170         nnodes = 10
171         cmodes = 2
172
173         u = ModRed.createUnitVector(1,  nnodes,  cmodes)
```

```
174             self . assertEqual (u [0,0],   1)
175             self . assertEqual (u [1,0],   0)
176             self . assertEqual (u [6,0],   1)
177             self . assertEqual (u.ColumnCount, 1)
178             self . assertEqual (u.RowCount, (nnodes*6) + cmodes)
179             # Checking if component mode positions are 0:
180             self . assertEqual (u[nnodes*6, 0], 0)
181             self . assertEqual (u[nnodes*6 + 1, 0], 0)
182
183             u = ModRed.createUnitVector(2, nnodes, cmodes)
184             self . assertEqual (u [0,0],   0)
185             self . assertEqual (u [1,0],   1)
186             self . assertEqual (u [7,0],   1)
187             self . assertEqual (u.ColumnCount, 1)
188             self . assertEqual (u.RowCount, (nnodes*6) + cmodes)
189             # Checking if component mode positions are 0:
190             self . assertEqual (u[nnodes*6, 0], 0)
191             self . assertEqual (u[nnodes*6 + 1, 0], 0)
192
193         def test_extractPartitions ( self ):
194             def array(*x): return sys_array [ float ](x)  # Helper function to create custom
              Math.NET matrices
195
196             mat = self . _indexMat_
197             nENodes = 1
198             dofs = 3
199             parts = ModRed. extractPartitions (mat, nENodes, dofs)
200             parts_ee = parts [0]
201             parts_ei = parts [1]
202             parts_ie = parts [2]
203             parts_ii = parts [3]
204
205             corr_ee = la.Double.Matrix.Build.DenseOfRowArrays(
206                 array (11, 12, 13),
207                 array (21, 22, 23),
208                 array (31, 32, 33)
209             )
210             self . assertEqual ( parts_ee , corr_ee )
211
212             corr_ei = la.Double.Matrix.Build.DenseOfRowArrays(
213                 array (14, 15, 16, 17, 18, 19),
214                 array (24, 25, 26, 27, 28, 29),
215                 array (34, 35, 36, 37, 38, 39)
216             )
217             self . assertEqual ( parts_ei , corr_ei )
218
219             corr_ie = la.Double.Matrix.Build.DenseOfRowArrays(
220                 array (41, 42, 43),
221                 array (51, 52, 53),
222                 array (61, 62, 63),
```

```
223                 array (71,  72,  73),
224                 array (81,  82,  83),
225                 array (91,  92,  93),
226             )
227             self . assertEqual ( parts_ie ,   corr_ie )
228
229             corr_ii   = la . Double.Matrix . Build . DenseOfRowArrays(
230                 array (44,  45,  46,  47,  48,  49),
231                 array (54,  55,  56,  57,  58,  59),
232                 array (64,  65,  66,  67,  68,  69),
233                 array (74,  75,  76,  77,  78,  79),
234                 array (84,  85,  86,  87,  88,  89),
235                 array (94,  95,  96,  97,  98,  99)
236             )
237             self . assertEqual (  parts_ii  ,   corr_ii  )
238
239         def    test_extractPartitions_FedemMatrix   ( self ) :
240             with  open( self .  _resourcesFolder   + "\\beam.nas" + "\\Fedem" +
          "\\ M_full_partitioned . p")  as   file:
241                 M_full  =  pickle . load( file )
242             mat = ModRed. extractPartitions (M_full ,  4,  6)
243
244             M_ee = mat[0]
245             M_ei = mat[1]
246             M_ie = mat[2]
247             M_ii  = mat[3]
248
249             self . assertEqual (M_ee.RowCount, 24)
250             self . assertEqual (M_ee.ColumnCount, 24)
251             self . assertEqual (M_ee [0,0],   9.775000E−002)
252             self . assertEqual (M_ee [0,1],   0.0)
253             self . assertEqual (M_ee [1,0],   0.0)
254             self . assertEqual (M_ee [23,0],   0.0)
255             self . assertEqual (M_ee [0,23],   0.0)
256             self . assertAlmostEqual (M_ee[23,23],  4.072916E−006)
257
258             self . assertEqual (M_ei.RowCount, 24)
259             self . assertEqual (M_ei.ColumnCount, 606)
260             self . assertEqual (M_ei [0,0],   M_full [0,   24])
261             self . assertEqual (M_ei[0,   605],   M_full [0,   629])
262             self . assertEqual (M_ei[23,   0],   M_full [23,   24])
263             self . assertEqual (M_ei[23,   605],   M_full [23,   629])
264
265             self . assertEqual (M_ie.RowCount, 606)
266             self . assertEqual (M_ie.ColumnCount, 24)
267             self . assertEqual (M_ie [0,0],   M_full [24,   0])
268             self . assertEqual (M_ie[0,   23],   M_full [24,   23])
269             self . assertEqual (M_ie[605,   0],   M_full [629,   0])
270             self . assertEqual (M_ie[605,   23],   M_full [629,   23])
271
```

```
272            self . assertEqual (M_ii.RowCount, 606)
273            self . assertEqual (M_ii.ColumnCount, 606)
274            self . assertEqual (M_ii [0,0],   M_full [24,  24])
275            self . assertEqual (M_ii [0,  605],  M_full [24,  629])
276            self . assertEqual (M_ii [605,  0],  M_full [629,  24])
277            self . assertEqual (M_ii [605,  605],  M_full [629,  629])
278
279
280
281        def  test_massMatrix_correct_mass_beam ( self ):
282            """
283            Testing  if  the  mass matrix  is  correct  by applying
284            u_trans ^T * M * u_trans  = m_tot
285            """
286            #  Validating  M_full  from Fedem
287            M_full  = ModRed.readFedemMatrix(self._resourcesFolder  + "\\beam.nas" +
         "\\Fedem" + "\\M_full.res")
288            correct_mass  = 3.12800E+01  # Taken from fedem_reducer. res .  In  kg
289            for  i  in  [1,  2,  3]:
290                u = ModRed.createUnitVector(i ,  105)
291                m = u.Transpose() . Multiply (M_full) . Multiply (u) [0,0]
292                self . assertAlmostEqual( correct_mass ,  m)
293
294            #  Testing  mass matrix  from ANSYS
295            M_full = ModRed.readMMFMatrix(self._resourcesFolder + "\\beam.nas" + "\\Ansys"
         + "\\M_full.mmf", "full")
296            for  i  in  [1,  2,  3]:
297                u  = ModRed.createUnitVector(i ,  105)
298                m = u.Transpose() . Multiply (M_full) . Multiply (u) [0,0]
299                self . assertAlmostEqual( correct_mass ,  m)
300
301            #  Testing  the  lumped mass matrix  from ANSYS
302            M_full  = ModRed.readMMFMatrix(self._resourcesFolder + "\\beam.nas" + "\\Ansys"
         + "\\M_full_lumped.mmf", "full")
303            for  i  in  [1,  2,  3]:
304                u  = ModRed.createUnitVector(i ,  105)
305                m = u.Transpose() . Multiply (M_full) . Multiply (u) [0,0]
306                print (m)
307                self . assertAlmostEqual( correct_mass ,  m)
308
309
310
311        def   test_partitionMatrix_correct_mass_beam   ( self ):
312            """
313            Testing  if  the  mass is  unchanged after   partitionMatrix   method
314            """
315            M_full  = ModRed.readMMFMatrix(self._resourcesFolder + "\\beam.nas" + "\\Ansys"
         + "\\M_full.mmf", "full")
316            eNodes = [1,  2,  3,  4]
317            iNodes = [i  for  i  in  range (5,  106)]
```

```
318            dict  = ModRed.readMappingFile(self._resourcesFolder  + "\\beam.nas" + "\\Ansys"
         + "\\M_full.mapping")
319            M_full  = ModRed.partitionMatrix(M_full,  eNodes, iNodes,  dict ,  6)
320
321            correct_mass  = 3.12800E+01  # Taken from fedem_reducer.res .  In  kg
322            for  i  in  [1,  2,  3]:
323                u = ModRed.createUnitVector(i ,  105)
324                m = u.Transpose().Multiply(M_full).Multiply(u)[0,0]
325                print(m)
326                self.assertAlmostEqual(correct_mass ,  m)
327
328
329
330
331        def   test_partitionMatrix_smallMatrices  (self):
332            """
333            Testing  the  partitionMatrix  method with  small  matrices
334            """
335            def  array(*x):  return  sys_array[float](x)  # Helper  function  to  create  custom
         Math.NET matrices
336            dict  = {                # dof  = equation  − 1
337                1:  2,
338                2:  3,
339                3:  1
340            }
341            # Remember: 3 DOFs per node
342            eNodes = [1]
343            iNodes =  [2,  3]
344            partitioned  = ModRed.partitionMatrix(self._indexMatSmall_,  eNodes, iNodes,
         dict ,  1,  sorting="dof")
345            correct  = la.Double.Matrix.Build.DenseOfRowArrays(
346                array(22,  23,  21),
347                array(32,  33,  31),
348                array(12,  13,  11)
349            )
350            self.assertEqual(correct ,  partitioned)
351            for  row  in  range(0,  correct.RowCount):
352                for  col  in  range(0,  correct.ColumnCount):
353                    self.assertEqual(partitioned[row,  col],  correct[row,  col])
354
355
356
357        def   test_partitionMatrix_5x5  (self):
358            #
359            # Testing  of  partitioning  a custom 5x5  matrix
360            #
361            def  array(*x):  return  sys_array[float](x)  # Helper  function  to  create  custom
         Math.NET matrices
362            dict  = {          # dict[dof]  = eq
363                1:  3,
```

```
364                 2: 4,
365                 3: 5,
366                 4: 1,
367                 5: 2
368             }
369             eNodes = [2, 5]
370             iNodes = [1, 3, 4]
371             partitioned = ModRed.partitionMatrix( self ._indexMat5x5_, eNodes, iNodes, dict ,
        1, sorting =”equation”)
372             print ( partitioned )
373
374
375         def test_partitionMatrix_9x9 ( self ):
376             #
377             # Testing with 9x9 matrix
378             #
379             def array(*x): return sys_array [ float ](x) # Helper function to create custom
        Math.NET matrices
380             dict = {
381                 1: 4,
382                 2: 5,
383                 3: 6,
384                 4: 7,
385                 5: 8,
386                 6: 9,
387                 7: 1,
388                 8: 2,
389                 9: 3
390             }
391             eNodes = [1]
392             iNodes = [2, 3]
393             partitioned = ModRed.partitionMatrix( self ._indexMat_, eNodes, iNodes, dict , 3,
        sorting =”dof”)
394             ”””
395             tmp = la .Double.Matrix.Build.DenseOfRowArrays(   # Moving rows
396                 array (41, 42, 43, 44, 45, 46, 47, 48, 49),
397                 array (51, 52, 53, 54, 55, 56, 57, 58, 59),
398                 array (61, 62, 63, 64, 65, 66, 67, 68, 69),
399                 array (71, 72, 73, 74, 75, 76, 77, 78, 79),
400                 array (81, 82, 83, 84, 85, 86, 87, 88, 89),
401                 array (91, 92, 93, 94, 95, 96, 97, 98, 99).
402                 array (11, 12, 13, 14, 15, 16, 17, 18, 19),
403                 array (21, 22, 23, 24, 25, 26, 27, 28, 29),
404                 array (31, 32, 33, 34, 35, 36, 37, 38, 39)
405             )
406             ”””
407             correct = la .Double.Matrix.Build.DenseOfRowArrays(   # Moving cols
408                 array (44, 45, 46, 47, 48, 49, 41, 42, 43),
409                 array (54, 55, 56, 57, 58, 59, 51, 52, 53),
410                 array (64, 65, 66, 67, 68, 69, 61, 62, 63),
```

```
411                 array (74,  75,  76,  77,  78,  79,  71,  72,  73),
412                 array (84,  85,  86,  87,  88,  89,  81,  82,  83),
413                 array (94,  95,  96,  97,  98,  99,  91,  92,  93),
414                 array (14,  15,  16,  17,  18,  19,  11,  12,  13),
415                 array (24,  25,  26,  27,  28,  29,  21,  22,  23),
416                 array (34,  35,  36,  37,  38,  39,  31,  32,  33)
417             )
418             for  row  in  range (0,   correct .RowCount):
419                 for  col  in  range (0,   correct .ColumnCount):
420                     self . assertEqual ( partitioned [row,  col ],   correct [row,  col ])
421
422
423         def  test_dofs2Eqs ( self ):
424             dofs  =  [1,  2,  3]
425             dict  = {
426                 1:  2,
427                 2:  3,
428                 3:  1
429             }
430             correct  =  [2,  3,  1]
431             eqs  = ModRed.dofs2Eqs(dofs, dict )
432             self . assertEqual (eqs ,   correct )
433             self . assertEqual ( len (dofs ) ,   len (eqs ) )
434
435
436             with  open( self . _resourcesFolder   + ”\\beam.nas” + ”\\Ansys” +
          ”\\M_full.mapping.p”)  as   file:
437                 dict  =  pickle . load( file )
438             dofs  = range (1,  631)
439             eqs  = ModRed.dofs2Eqs(dofs, dict )
440
441             self . assertEqual ( len (dofs ) ,   len (eqs ) )
442             self . assertEqual (eqs [0],   625)          # dof  1  coupled  to  equation  624
443             self . assertEqual (eqs [1],   626)
444             self . assertEqual (eqs [282],   1)
445
446
447         def  test_nodes2Dofs ( self ):
448             ”””
449             # Example:
450             # eNodes = [1,  5,  10]  in  elements  with  6  dofs
451             # => eDofs = [1,  2,  3,  4,  5,  6,  25,  26,  27,  28,  29,  30,  55,  56,  57,  58,  59,  60]
452             ”””
453             eNodes = [1]
454             dofs  = 6
455             correct  =  [1,  2,  3,  4,  5,  6]
456
457             eDofs = ModRed.nodes2Dofs(eNodes, dofs)
458             self . assertEqual (eDofs,   correct )
459
```

```
460         eNodes = [1,  5,  10]
461         correct  = [1,  2,  3,  4,  5,  6,  25,  26,  27,  28,  29,  30,  55,  56,  57,  58,  59,  60]
462         eDofs = ModRed.nodes2Dofs(eNodes, dofs)
463         self . assertEqual (eDofs,  correct )
464
465
466         eNodes = [1,  3]
467         dofs  = 2
468         correct  = [1,  2,  5,  6]
469         eDofs = ModRed.nodes2Dofs(eNodes, dofs)
470         self . assertEqual (eDofs,  correct )
471
472
473
474
475     def  test_partitionMatrix_difference_twoQUAD4 ( self ) :
476         """
477         Testing  if  Fedem and ANSYS matrices are  partitioned   correctly .
478         Using the  partitioned  Fedem matrix as  reference ,  as  this  is  proven  to  yield
479         the  identical  gravity  vector  as  Fedem calculates .
480         """
481         dict_fedem  = ModRed.readMappingFileFedem(self._resourcesFolder +
        "\\twoQUAD4" + "\\Fedem" + "\\MEQN.res")
482         dict_ansys  = ModRed.readMappingFile(self. _resourcesFolder  + "\\twoQUAD4" +
        "\\Ansys" + "\\M_full_lumped.mapping")
483         M_full_fedem = ModRed.readFedemMatrix(self. _resourcesFolder  + "\\twoQUAD4" +
        "\\Fedem" + "\\M_full.res")
484         M_full_ansys  = ModRed.readMMFMatrix(self._resourcesFolder + "\\twoQUAD4" +
        "\\Ansys" + "\\M_full_lumped.mmf", "full")
485
486         eNodes = [1,  3]
487         iNodes =  [2,  4,  5,  6]
488         fedem = ModRed.partitionMatrix(M_full_fedem,  eNodes, iNodes,  dict_fedem ,  6,
        sorting ="equation")
489         ansys  = ModRed.partitionMatrix( M_full_ansys ,  eNodes, iNodes,  dict_ansys ,  6,
        sorting ="equation")
490
491         for  row  in  range (0,  ansys . RowCount):
492             for  col  in  range (0,  ansys . ColumnCount):
493                 if  row==col:
494                     print ("r:  "  +  str (row)  + ",\ tc:  "  +  str (col)  + ",\ tfedem:  "  +
        str (fedem[row,  col ])  + ",\ t\ tansys:  "  +  str (ansys[row,  col ])  )
495                     self . assertAlmostEqual (fedem[row,  col ],  ansys [row,  col ],  places =1)
496         self . fail ("Not  correct ")
497
498
499
500
501     def  test_massMatrix_diagonality  ( self ) :
502         #
```

```
503            # Verifying that a lumped matrix is diagonal
504            #
505            mat = ModRed.readMMFMatrix(self._resourcesFolder + "\\beam.nas" + "\\Ansys" +
         "\\M_full_lumped.mmf", "full")
506            for row in range(0, mat.RowCount):
507                for col in range(0, mat.ColumnCount):
508                    if row == col:
509                        self.assertTrue(mat[row, col] != 0.0 )
510                    else:
511                        self.assertTrue(mat[row, col] == 0.0)
512
513
514
515        def test_partitionMatrix_twoQUAD4_Ansys(self):
516            #
517            # Testing with mass matrix from twoQUAD4.
518            #
519            dict = ModRed.readMappingFileFedem(self._resourcesFolder + "\\twoQUAD4" +
         "\\Fedem" + "\\MEQN.res")
520            M_full = ModRed.readFedemMatrix(self._resourcesFolder + "\\twoQUAD4" +
         "\\Fedem" + "\\M_full.res")
521
522            eNodes = [1, 3]
523            iNodes = [2, 4, 5, 6]
524            mat = ModRed.partitionMatrix(M_full, eNodes, iNodes, dict, 6)
525            # Checking if diagonal:
526            for row in range(0, mat.RowCount):
527                for col in range(0, mat.ColumnCount):
528                    if row == col:
529                        self.assertTrue(mat[row, col] != 0.0 )
530                    else:
531                        self.assertTrue(mat[row, col] == 0.0)
532
533            # It seems that some of the precision is lost in Fedem when comparing M_full
         and M_ee from Fedem.
534            # Therefore the assertAlmostEqual
535            self.assertAlmostEqual(mat[0,0], 3.910000E+001)
536            self.assertAlmostEqual(mat[1,1], 3.910000E+001)
537            self.assertAlmostEqual(mat[2,2], 3.910000E+001)
538            self.assertAlmostEqual(mat[3,3], 2.782139E-001, places=6)
539            self.assertAlmostEqual(mat[5,5], 6.516665E-001, places=6)
540            self.assertAlmostEqual(mat[11,11], 6.516665E-001, places=6)
541
542
543        def test_partitionMatrix_twoQUAD4_Fedem(self):
544            #
545            # Testing with mass matrix + dictionary from ANSYS. Compared to M_ee from
         Fedem
546            #
```

```
547            dict  = ModRed.readMappingFile(self. _resourcesFolder  + "\\twoQUAD4" +
         "\\Ansys" + "\\M_full.mapping")
548            M_full  = ModRed.readMMFMatrix(self._resourcesFolder + "\\twoQUAD4" +
         "\\Ansys" + "\\M_full_lumped.mmf", "full")
549
550            eNodes = [1,  3]
551            iNodes =  [2,  4,  5,  6]
552            mat = ModRed.partitionMatrix(M_full,  eNodes, iNodes,  dict ,  6)
553            correct  = ModRed.readFedemVector(self. _resourcesFolder  + "\\twoQUAD4" +
         "\\Fedem" + "\\M_ee.res")
554
555            # Checking if  diagonal:
556            for  row in  range(0,  mat.RowCount):
557                for  col  in  range(0,  mat.ColumnCount):
558                    if  row == col:
559                        self . assertTrue (mat[row,  col ]  !=  0.0  )
560                    else:
561                        self . assertTrue (mat[row,  col ]  == 0.0)
562
563            #  It  seems  that  some of  the   precision   is   lost   in  Fedem when comparing M_full
         and M_ee from Fedem.
564            #  Therefore  the  assertAlmostEqual
565            self . assertAlmostEqual(mat [0,0],   3.910000E+001, places=2)
566            self . assertAlmostEqual(mat [1,1],   3.910000E+001, places=2)
567            self . assertAlmostEqual(mat [2,2],   3.910000E+001, places=2)
568            self . assertAlmostEqual(mat [3,3],   2.782139E−001, places=2)
569            self . assertAlmostEqual(mat [5,5],   6.516665E−001, places=2)
570            self . assertAlmostEqual(mat [9,9],   2.782139E−001, places=5)
571            self . assertAlmostEqual(mat [11,11],  6.516665E−001, places=2)
572
573
574        def   test_partitionMatrix_Fedem_beam ( self ) :
575            #
576            #  Testing  with  mass matrix  from beam.nas.  Using the  Fedem matrices,
577            # because  here  we know the  resulting   matrix
578            #
579            dict  = ModRed.readMappingFileFedem(self._resourcesFolder + "\\beam.nas" +
         "\\Fedem" + "\\MEQN.res")
580            M_full  = ModRed.readFedemMatrix(self. _resourcesFolder  + "\\beam.nas" +
         "\\Fedem" + "\\M_full.res")
581
582            eNodes = [1,  2,  3,  4]
583            iNodes = [ i  for  i  in  range(5,  106)]
584            mat = ModRed.partitionMatrix(M_full,  eNodes, iNodes,  dict ,  6)
585            # Checking if  diagonal:
586            for  row in  range(0,  mat.RowCount):
587                for  col  in  range(0,  mat.ColumnCount):
588                    if  row == col:
589                        self . assertTrue (mat[row,  col ]  !=  0.0  )
590                    else:
```

```
591                         self . assertTrue (mat[row,  col ]  == 0.0)
592
593             # It seems that some of the precision is lost in Fedem when comparing M_full
            and M_ee from Fedem.
594             # Therefore the assertAlmostEqual
595             self . assertAlmostEqual(mat [0,0],  9.775000E−002)
596             self . assertEqual (mat [1,0],  0.0)
597             self . assertEqual (mat [0,1],  0.0)
598             self . assertEqual (mat [1,1],  9.775000E−002)
599             self . assertAlmostEqual(mat [3,3],  1.982601E−006)
600             self . assertAlmostEqual(mat [4,4],  3.048196E−006)
601             self . assertAlmostEqual(mat [5,5],  4.07292E−06)
602
603             self . assertAlmostEqual(mat [11,11],  4.072916E−006)
604             self . assertAlmostEqual(mat[15,  15],  1.982601E−006)
605             self . assertAlmostEqual(mat[22,  22],  3.048196E−006)
606             self . assertAlmostEqual(mat[23,  23],  4.072916E−006)
607
608             self . assertEqual (mat[24,  24],  1.95500E−01)
609             self . assertEqual (mat[629,  629],  1.62917E−05)
610
611
612         def  test_partitionMatrix_Fedem_unsrtENodes ( self ):
613
614             # beam.nas model from Fedem, with selected eNodes = [22,  28,  6,  44]
615
616             dict  = ModRed.readMappingFileFedem(self._resourcesFolder + "\\beam.nas" +
            "\\Fedem" + "\\unsrtENodes" + "\\MEQN.res")
617             M_full  = ModRed.readFedemMatrix(self._resourcesFolder  + "\\beam.nas" +
            "\\Fedem" + "\\unsrtENodes" + "\\M_full.res")        # NOTE: The full mass matrix is
            not the same
618
619             eNodes = [6,  22,  28,  44]
620             iNodes = [ i  for  i  in  range (1,  106)]
621             for  e  in  eNodes:
622                 iNodes.remove(e)
623
624             mat = ModRed.partitionMatrix(M_full,  eNodes, iNodes,  dict ,  6)
625             # Checking if diagonal:
626             for  row  in  range (0,  mat.RowCount):
627                 for  col  in  range (0,  mat.ColumnCount):
628                     if  row == col:
629                         self . assertTrue (mat[row,  col ]  != 0.0  )
630                     else:
631                         self . assertTrue (mat[row,  col ]  == 0.0)
632
633             # Checking entries in M_ee:
634             self . assertAlmostEqual(mat [0,0],  1.955000E−001)
635             self . assertAlmostEqual(mat [3,3],  3.965202E−006)
636             self . assertAlmostEqual(mat [6,6],  1.955000E−001)
```

```
637            self.assertAlmostEqual(mat[16,16], 6.096392E-006)
638
639
640
641        def test_partitionMatrix_beam_Mee(self):
642            # Testing with M_full from beam.nas (medium model)
643            mapping = ModRed.readMappingFileFedem(self._resourcesFolder + "\\beam.nas" +
           "\\Fedem" + "\\MEQN.res")
644            M_full = ModRed.readFedemMatrix(self._resourcesFolder + "\\beam.nas" +
           "\\Fedem" + "\\M_full.res")
645            eNodes = [1, 2, 3, 4]
646            iNodes = [i for i in range(5, 106)]
647            M_full_partitioned = ModRed.partitionMatrix(M_full, eNodes, iNodes, mapping, 6)
648            # Using M_ee from Fedem as reference
649            M_ee = ModRed.readFedemVector(self._resourcesFolder + "\\beam.nas" +
           "\\Fedem" + "\\M_ee.res")
650
651            for row in range(0, M_ee.RowCount):
652                for col in range(0, M_ee.ColumnCount):
653                    self.assertAlmostEqual(M_full_partitioned[row, col], M_ee[row, col])
654
655
656
657
658        def test_partitionMatrix_noDict(self):
659            """
660            Testing the partitionMatrix method with dictionary dof = equation
661            """
662            def array(*x): return sys_array[float](x)  # Helper function to create custom
           Math.NET matrices
663            # Dictionary coupling dof to equation:
664            # dict[dof] = eq
665            noDict = {       # dof = equation
666                1: 1,
667                2: 2,
668                3: 3,
669                4: 4,
670                5: 5,
671                6: 6,
672                7: 7,
673                8: 8,
674                9: 9
675            }
676            # Remember: 3 DOFs per node
677            eNodes = [1]
678            iNodes = [2, 3]
679
680            partitioned = ModRed.partitionMatrix(self._mat_, eNodes, iNodes, noDict, 3,
           sorting="dof")
681            # Dictionary with dof = equation and eNodes, iNodes in ascending order
```

```
682             # should return the same matrix
683             self . assertEqual ( self . _mat_,  partitioned )
684
685             partitioned = ModRed.partitionMatrix( self . _mat_, eNodes, iNodes, noDict, 3,
        sorting ="dof")
686             # Testing first if partitionMatrix works with no dictionary mapping
687
688             correct = la . Double.Matrix . Build . DenseOfRowArrays( # Moving rows
689                 array (31, 32, 33, 28, 29, 30, 34, 35, 36),
690                 array (40, 41, 42, 37, 38, 39, 43, 44, 45),
691                 array (49, 50, 51, 46, 47, 48, 52, 53, 54),
692                 array (4, 5, 6, 1, 2, 3, 7, 8, 9),
693                 array (13, 14, 15, 10, 11, 12, 16, 17, 18),
694                 array (22, 23, 24, 19, 20, 21, 25, 26, 27),
695                 array (58, 59, 60, 55, 56, 57, 61, 62, 63),
696                 array (67, 68, 69, 64, 65, 66, 70, 71, 72),
697                 array (76, 77, 78, 73, 74, 75, 79, 80, 81)
698             )
699
700             eNodes = [2]
701             iNodes = [1, 3]
702             partitioned = ModRed.partitionMatrix( self . _mat_, eNodes, iNodes, noDict, 3,
        sorting ="dof")
703             self . assertEqual ( partitioned , correct )
704
705
706
707
708         def test_mathNET2list ( self ) :
709             list = ModRed.mathNET2list(self._mat_)
710             correct_list = [1.0, 10.0, 19.0, 28.0]
711             wrong_list = [1.0, 2.0, 3.0, 4.0]          # Should be returned column−first
712
713             print ("my list: " + str ( list [0:4 ]) )
714             self . assertEqual ( list [0:4 ],  correct_list )
715             self . assertNotEqual ( list [0:4 ], wrong_list )
716             self . assertEqual (len( list ), 81)        # 9x9 matrix
717
718
719         def test_readMMFMatrix_red( self ) :
720             M_red_path = self . _resourcesFolder + "\\beam.nas" + "\\Ansys" + "\\M_red.mmf"
721             mat = ModRed.readMMFMatrix(M_red_path, "sub")
722
723             self . assertEqual (mat.RowCount, 26)
724             self . assertEqual (mat.ColumnCount, 26)
725
726             self . assertEqual (mat [0,0], 9.037402289784589E+00)
727             self . assertEqual (mat [0,2], 0)
728             self . assertEqual (mat [2,0], 0)
729
```

```
730            self . assertEqual (mat [0,5],   4.528581487170890E−05)
731            self . assertEqual (mat [5,0],   4.528581487170890E−05)
732
733            self . assertEqual (mat [25,25],   1.0)
734
735            M_red_path =  self . _resourcesFolder  + "\\plate . nas"  + "\\Ansys" + "\\M_red.mmf"
736            mat = ModRed.readMMFMatrix(M_red_path, "sub")
737
738            self . assertEqual (mat.RowCount, 26)
739            self . assertEqual (mat.ColumnCount, 26)
740
741            self . assertEqual (mat [0,0],   5.871600155722680E−01)
742            self . assertEqual (mat [0,2],   4.381258188504480E−01)
743            self . assertEqual (mat[24,  20],  2.027417791055110E−17)
744
745
746
747
748
749
750        def  test_readMMFMatrix_full( self ):
751            path  =  self . _resourcesFolder   + "\\beam.nas" + "\\Ansys" + "\\M_full.mmf"
752            mat = ModRed.readMMFMatrix(path, "full")
753
754            # Checking if  dimensions  are   correct
755            self . assertEqual (mat.RowCount, 630)
756            self . assertEqual (mat.ColumnCount, 630)
757
758            # Checking some values  at  the  boundaries:
759            self . assertEqual (mat [0,0],   8.688888888888170E−02)
760            self . assertEqual (mat[0,  629],  0)
761            self . assertEqual (mat[629,  0],  0)
762            self . assertEqual (mat[629,  629],  1.448148148148030E−16)
763
764            #  Testing  random values:
765            self . assertEqual (mat [23,23],   1.448148148148030E−16)
766            self . assertEqual (mat [285,243],  3.620370370370130E−07)
767            self . assertEqual (mat [291,291],  5.792592592592110E−06)
768
769
770
771
772        def  test_readMMFMatrix_full_isSymmetric( self ):
773            path  =  self . _resourcesFolder   + "\\beam.nas" + "\\Ansys" + "\\M_full.mmf"
774            mat = ModRed.readMMFMatrix(path, "full")
775
776            # Checking if  matrix  is  symmetric:
777            for  row  in  range (0,  mat.RowCount):
778                for  col  in  range (0,  mat.ColumnCount):
779                    self . assertEqual (mat[row,  col ],   mat[col ,  row])
```

```
780
781
782     def test_readMMFMatrix_full_correctMass ( self ):
783         path = self . _resourcesFolder + "\\beam.nas" + "\\Ansys" + "\\M_full.mmf"
784         mat = ModRed.readMMFMatrix(path, "full")
785         correct_mass = 3.12800E+01          # Taken from Fedem_reducer
786
787         for i in [1, 2, 3]:
788             u = ModRed.createUnitVector(i , 105)
789             m = u.Transpose() . Multiply (mat). Multiply (u) [0,0]
790             print (m)
791             self . assertAlmostEqual( correct_mass , m)
792
793
794     def test_readMMFMatrix_red_isSymmetric( self ):
795         path = self . _resourcesFolder + "\\beam.nas" + "\\Ansys" + "\\M_red.mmf"
796         mat = ModRed.readMMFMatrix(path, "sub")
797
798         # Checking if matrix is symmetric:
799         for row in range (0, mat.RowCount):
800             for col in range (0, mat.ColumnCount):
801                 self . assertEqual (mat[row, col ], mat[col , row])
802
803
804     def test_readFedemVector_red_correctMass ( self ):
805         path = self . _resourcesFolder + "\\beam.nas" + "\\Fedem" + "\\M_red.res"
806         mat = ModRed.readFedemVector(path)
807         print (mat)
808         correct_mass = 3.12800E+01
809
810         for i in [1, 2, 3]:
811             u = ModRed.createUnitVector(i , 4, 2)
812             m = u.Transpose() . Multiply (mat). Multiply (u) [0,0]
813             self . assertAlmostEqual( correct_mass , m, places=5)
814
815
816     def test_readMMFMatrix_red_correctMass( self ):
817         """
818         Applying unit translation in x, y, z direction and verifying it is the same
    mass
819         as computed in Fedem
820         """
821         path = self . _resourcesFolder + "\\beam.nas" + "\\Ansys" + "\\M_red.mmf"
822         mat = ModRed.readMMFMatrix(path, "sub")
823         correct_mass = 3.12800E+01
824
825         for i in [1, 2, 3]:
826             u = ModRed.createUnitVector(i , 4, 2)
827             m = u.Transpose() . Multiply (mat). Multiply (u) [0,0]
828             self . assertAlmostEqual( correct_mass , m, places=6)
```

```
829
830
831
832
833
834     """
835     # Commented out for being able to run all tests faster
836     def test_readMMFMatrix_largeFiles(self):
837         M_full_path = self._resourcesFolder + "\\lca.nas" + "\\M_full.mmf"    # Large
        file (230 000 lines)
838         M_full = ModRed.readMMFMatrix(M_full_path, "full")
839         self.assertEqual(M_full.RowCount, 35553)
840         self.assertEqual(M_full.ColumnCount, 35553)
841         self.assertEqual(M_full[0, 0], 6.837606621530309E-05)    # start
842         self.assertEqual(M_full[165, 12], 2.229360838501640E-05)    # random place
843         self.assertEqual(M_full[35552, 35552], 2.580374027760560E-07)  # end
844
845
846     def test_readMMFMatrix_largeFiles2(self):
847         M_full_path = self._resourcesFolder + "\\reactor" + "\\M_full.mmf"    # Large
        file (450 000 lines)
848         M_full = ModRed.readMMFMatrix(M_full_path, "full")
849         self.assertEqual(M_full.RowCount, 69438)
850         self.assertEqual(M_full.ColumnCount, 69438)
851         self.assertEqual(M_full[0, 0], 9.966062445579789E-05)    # start
852         self.assertEqual(M_full[26236, 26236], 7.549508902862550E-04)  # random place
853         self.assertEqual(M_full[69437, 69437], 8.306723475513540E-04)    # end
854     """
855
856     def test_readFedemMatrix(self):
857         path = self._resourcesFolder + "\\beam.nas" + "\\Fedem" + "\\M_full.res"
858         mat = ModRed.readFedemMatrix(path)
859         # Checking some entries
860         self.assertEqual(0.1955, mat[0,0])
861         self.assertEqual(mat[0,1], 0.000)
862         self.assertEqual(mat[1,1], 0.1955)
863         self.assertEqual(mat[202, 202], 6.09639E-06)
864         self.assertEqual(mat[605, 605], 1.62917E-05)
865         self.assertEqual(mat[606, 606], 9.77500E-02)
866         self.assertEqual(mat[629, 629], 4.07292E-06)
867
868     def test_readFedemVector_twoQUAD4(self):
869         path = self._resourcesFolder + "\\twoQUAD4" + "\\Fedem" + "\\gravVec.res"
870         mat = ModRed.readFedemVector(path)
871
872         self.assertEqual(mat[0,0], 1.564000E+002)
873         self.assertEqual(mat[0,1], -3.556089E+001)
874         self.assertEqual(mat[13,2], -9.710526E-015)
875         self.saveObject(mat, "gravVec")
876
```

```
877
878     def test_readFedemVector ( self ):
879         path = self . _resourcesFolder + "\\beam.nas" + "\\Fedem" + "\\Bmat.res"
880         mat = ModRed.readFedemVector(path)
881
882         self . assertEqual (mat [0,0], 2.270180E−001)
883         self . assertEqual (mat [0,1], 8.015493E−002)
884         self . assertEqual (mat [0,9], 0.0)
885         self . assertEqual (mat [0,17], −3.023801E−004)
886         self . assertEqual (mat [0,23], −7.732552E−003)
887         self . assertEqual (mat [601,0], 1.497216E−002)
888         self . assertEqual (mat [604,0], 0.0)
889         self . assertEqual (mat[605, 23], −2.384053E−002)
890         self . assertEqual (mat [605,23], −2.384053E−002)
891         self . assertEqual (mat [604,9], −2.839266E−004)
892         self . assertEqual (mat [605,9], 0.0)
893
894         self . assertEqual (mat [605,10], 4.054158E−003)
895         self . assertEqual (mat [605,11], −3.068604E−002)
896         self . assertEqual (mat [605,5], 4.054158E−003)
897         self . assertEqual (mat [605,1], 1.572045E−001)
898         self . assertEqual (mat [605,0], 5.364874E−001)
899
900         mat = ModRed.readFedemVector(self._resourcesFolder + "\\beam.nas" + "\\Fedem"
     + "\\phi_fedem.res")
901         self . assertEqual (0.0, mat [0,0])
902         self . assertEqual (6.392561E−016, mat[0,1])
903         self . assertEqual (−8.829041E−002, mat[2,0])
904         self . assertEqual (3.321254E−016, mat[605,0])
905         self . assertEqual (−5.843641E−014, mat[605,1])
906
907         vec = ModRed.readFedemVector(self._resourcesFolder + "\\beam.nas" + "\\Fedem"
     + "\\gravVec.res")
908         self . assertEqual (vec [0,0], 7.820000E+000)
909         self . assertEqual (vec [0,1], 1.297520E+001)
910         self . assertEqual (vec [25,0], 1.826227E−014)
911         self . assertEqual (vec [25,1], 4.104385E−014)
912         self . assertEqual (vec [25,2], −1.456509E−013)
913
914
915     def test_readMappingFileFedem( self ):
916         dict = ModRed.readMappingFileFedem(self._resourcesFolder + "\\beam.nas" +
     "\\Fedem" + "\\MEQN.res")
917
918         dof = 1
919         eq = 607
920         self . assertEqual (eq, dict [dof])
921
922         dof = 2
923         eq = 608
```

```
924             self . assertEqual (eq,  dict [dof ])
925
926             dof = 630
927             eq = 258
928             self . assertEqual (eq,  dict [dof ])
929
930
931
932         def  test_calculateGravityVector_Fedem_beam ( self ):
933
934             with open( self . _resourcesFolder  + "\\beam.nas" + "\\Fedem" + "\\gravVec.p") as
        file:
935                 correct  =  pickle . load( file )
936             with open( self . _resourcesFolder  + "\\beam.nas" + "\\Fedem" + "\\NOR.p") as file:
937                 NOR = pickle.load( file )
938             with open( self . _resourcesFolder  + "\\beam.nas" + "\\Fedem" +
        "\\identityMatrix . p")  as  file:
939                 ident  =  pickle . load( file )
940             B = ModRed.readFedemVector(self. _resourcesFolder  + "\\beam.nas" + "\\Fedem" +
        "\\Bmat.res")
941             CST = la.Double.SparseMatrix(630,  24)
942             CST.SetSubMatrix(0, 24,  0,  24,  ident )
943             CST.SetSubMatrix(24, 606,  0,  24,  B)
944             H = CST.Append(NOR)                 #  H = [CST NOR]
945
946             # Building  M_full:
947             M_full  = ModRed.readFedemMatrix(self._resourcesFolder  + "\\beam.nas" +
        "\\Fedem" + "\\M_full.res")
948             dict  = ModRed.readMappingFileFedem(self._resourcesFolder + "\\beam.nas" +
        "\\Fedem" + "\\MEQN.res")
949             eNodes = [1,  2,  3,  4]
950             iNodes = [ i  for  i  in  range (5,  106)]
951             M_full  = ModRed.partitionMatrix(M_full ,  eNodes, iNodes,  dict ,  6)
952
953             gravVec = ModRed.calculateGravityVector(M_full ,  H,  4,  101,  6)
954             print (gravVec)
955             for  row  in  range (0,  gravVec.RowCount):
956                 for  col  in  range (0,  gravVec.ColumnCount):
957                     self . assertAlmostEqual(gravVec[row,  col ],   correct [row,  col ],   places=5)
958
959
960
961
962
963         def  test_calculateGravityVectorReduced_beam ( self ):
964             with open( self . _resourcesFolder  + "\\beam.nas" + "\\Fedem" + "\\gravVec.p") as
        file:
965                 correct  =  pickle . load( file )
966
```

```
967             M_red = ModRed.readMMFMatrix(self._resourcesFolder + "\\beam.nas" + "\\Ansys" +
          "\\M_red.mmf", "sub")
968
969             gravVec = ModRed.calculateGravityVectorReduced(M_red, 4, 2)
970             diffs = []
971             for row in range(0, gravVec.RowCount):
972                 for col in range(0, gravVec.ColumnCount):
973                     diff = correct[row, col] − gravVec[row, col]
974                     diffs.append(diff)
975                     if (diff > 1.0):
976                         self.fail("Too large diff")
977
978
979         def test_calculateGravityVectorReduced_twoQUAD4(self):
980             with open(self._resourcesFolder + "\\twoQUAD4" + "\\Fedem" + "\\gravVec.p")
          as file:
981                 correct = pickle.load(file)
982
983             CST = ModRed.readMMFMatrix(self._resourcesFolder + "\\twoQUAD4" + "\\Ansys"
          + "\\CST.mmf", "full")
984             NOR = ModRed.readMMFMatrix(self._resourcesFolder + "\\twoQUAD4" + "\\Ansys"
          + "\\NOR.mmf", "full")
985             H = CST.Append(NOR)             # H = [CST NOR]
986
987             M_red = ModRed.readMMFMatrix(self._resourcesFolder + "\\twoQUAD4" +
          "\\Ansys" + "\\M_red.mmf", "sub")
988
989             gravVec = ModRed.calculateGravityVectorReduced(M_red, 2, 2)
990             print(gravVec)
991             diffs = []
992             for row in range(0, gravVec.RowCount):
993                 for col in range(0, gravVec.ColumnCount):
994                     diff = correct[row, col] − gravVec[row, col]
995                     diffs.append(diff)
996             print(diffs)
997             if (max(diffs)) > 20:
998                 self.fail("Too large diff")
999
1000
1001        def test_calculateGravityVectorReduced_plate(self):
1002            correct = ModRed.readFedemVector(self._resourcesFolder + "\\plate.nas" +
          "\\Fedem" + "\\gravVec.res")
1003
1004            M_red = ModRed.readMMFMatrix(self._resourcesFolder + "\\plate.nas" + "\\Ansys"
          + "\\M_red.mmf", "sub")
1005
1006            gravVec = ModRed.calculateGravityVectorReduced(M_red, 4, 2)
1007            print(gravVec)
1008            diffs = []
1009            for row in range(0, gravVec.RowCount):
```

```
1010                for col in range(0, gravVec.ColumnCount):
1011                    diff = correct[row, col] − gravVec[row, col]
1012                    diffs.append(diff)
1013
1014
1015            print(diffs)
1016            print(max(diffs))
1017            if (max(diffs)) > 20:
1018                self.fail("Too large diff")
1019
1020        def test_calculateGravityVector_Ansys_twoQUAD4(self):
1021
1022            with open(self._resourcesFolder + "\\twoQUAD4" + "\\Fedem" + "\\gravVec.p")
         as file:
1023                correct = pickle.load(file)
1024            M_full = ModRed.readMMFMatrix(self._resourcesFolder + "\\twoQUAD4" +
         "\\Ansys" + "\\M_full_lumped.mmf", "full")
1025            CST = ModRed.readMMFMatrix(self._resourcesFolder + "\\twoQUAD4" + "\\Ansys"
         + "\\CST.mmf", "full")
1026            NOR = ModRed.readMMFMatrix(self._resourcesFolder + "\\twoQUAD4" + "\\Ansys"
         + "\\NOR.mmf", "full")
1027            H = CST.Append(NOR)             # H = [CST NOR]
1028
1029            dict = ModRed.readMappingFile(self._resourcesFolder + "\\twoQUAD4" +
         "\\Ansys" + "\\M_full_lumped.mapping")
1030            eNodes = [1, 3]
1031            iNodes = [2, 4, 5, 6]
1032            M_full = ModRed.partitionMatrix(M_full, eNodes, iNodes, dict, 6)
1033
1034            gravVec = ModRed.calculateGravityVector(M_full, H, 2, 4, 6)
1035            print(" calculated: " + str(gravVec))
1036            print(" correct: " + str(correct))
1037            for row in range(0, gravVec.RowCount):
1038                for col in range(0, gravVec.ColumnCount):
1039                    self.assertAlmostEqual(gravVec[row, col], correct[row, col], places=1)
1040
1041
1042        def test_node2dof(self):
1043            dofDict = {
1044                "UX": 1,
1045                "UY": 2,
1046                "UZ": 3,
1047                "ROTX": 4,
1048                "ROTY": 5,
1049                "ROTZ": 6
1050            }
1051
1052            node = 1
1053            dofText = "UX"
1054            correctDof = 1
```

```
1055            dof = ModRed.node2dof(node, dofDict, 6, dofText)
1056            self . assertEqual (dof, correctDof)
1057
1058            node = 1
1059            dofText = "UY"
1060            correctDof = 2
1061            dof = ModRed.node2dof(node, dofDict, 6, dofText)
1062            self . assertEqual (dof, correctDof)
1063
1064            node = 1
1065            dofText = "UZ"
1066            correctDof = 3
1067            dof = ModRed.node2dof(node, dofDict, 6, dofText)
1068            self . assertEqual (dof, correctDof)
1069
1070            node = 1
1071            dofText = "ROTX"
1072            correctDof = 4
1073            dof = ModRed.node2dof(node, dofDict, 6, dofText)
1074            self . assertEqual (dof, correctDof)
1075
1076            node = 2
1077            dofText = "UY"
1078            correctDof = 8
1079            dof = ModRed.node2dof(node, dofDict, 6, dofText)
1080            self . assertEqual (dof, correctDof)
1081
1082            node = 3
1083            dofText = "UZ"
1084            correctDof = 15
1085            dof = ModRed.node2dof(node, dofDict, 6, dofText)
1086            self . assertEqual (dof, correctDof)
1087
1088
1089
1090        def test_calculateGravityVector_Fedem_twoQUAD4(self):
1091            with open( self . _resourcesFolder + "\\twoQUAD4" + "\\Fedem" + "\\gravVec.p")
         as file:
1092                correct = pickle . load( file )
1093            M_full = ModRed.readFedemMatrix(self._resourcesFolder + "\\twoQUAD4" +
         "\\Fedem" + "\\M_full.res")
1094            ident = la . Double.SparseMatrix. CreateIdentity (12)
1095            B = ModRed.readFedemVector(self._resourcesFolder + "\\twoQUAD4" + "\\Fedem"
         + "\\BMAT.res")
1096            nullMat = la . Double.SparseMatrix (12, 2)
1097            phi = ModRed.readFedemVector(self._resourcesFolder + "\\twoQUAD4" +
         "\\Fedem" + "\\phi.res")
1098            CST = la . Double.SparseMatrix (36, 12)
1099            CST.SetSubMatrix(0, 12, 0, 12, ident)
1100            CST.SetSubMatrix(12, 24, 0, 12, B)
```

```
1101            NOR = la.Double.SparseMatrix(36,  2)
1102            NOR.SetSubMatrix(0, 12,  0,  2,  nullMat)
1103            NOR.SetSubMatrix(12, 24, 0,  2,  phi)
1104            H = CST.Append(NOR)                # H = [CST NOR]
1105
1106            dict  = ModRed.readMappingFileFedem(self._resourcesFolder + "\\twoQUAD4" +
        "\\Fedem" + "\\MEQN.res")
1107            eNodes = [1,  3]
1108            iNodes = [2,  4,  5,  6]
1109            M_full  = ModRed.partitionMatrix(M_full,  eNodes, iNodes,  dict ,  6)
1110
1111
1112            gravVec = ModRed.calculateGravityVector(M_full,  H,  2,  4,  6)
1113            print (gravVec)
1114            for  row  in  range(0,  gravVec.RowCount):
1115                for  col  in  range(0,  gravVec.ColumnCount):
1116                    self . assertAlmostEqual(gravVec[row,  col ],   correct [row,  col ],   places=4)
1117
1118
1119
1120
1121        def   test_calculateGravityVector_Fedem_unsortedENodes ( self ):
1122            """
1123            Gravity  vector  with  eNodes = [22,  28,  6,  44]
1124            Using  the  beam.nas model
1125            """
1126            correct  = ModRed.readFedemVector(self. _resourcesFolder  + "\\beam.nas" +
        "\\Fedem" + "\\unsrtENodes" + "\\gravVec.res")
1127
1128            B = ModRed.readFedemVector(self. _resourcesFolder  + "\\beam.nas" + "\\Fedem" +
        "\unsrtENodes" + "\\Bmat.res")
1129            ident  =  la . Double.SparseMatrix(24) .  CreateIdentity  (24)
1130            CST = la.Double.SparseMatrix(630,  24)
1131            CST.SetSubMatrix(0, 24,  0,  24,  ident )
1132            CST.SetSubMatrix(24, 606,  0,  24,  B)
1133
1134            phi  = ModRed.readFedemVector(self. _resourcesFolder  + "\\beam.nas" + "\\Fedem"
        + "\unsrtENodes" + "\\phi.res")
1135            nullMat  =  la . Double.SparseMatrix(24,  2)
1136            NOR = la.Double.SparseMatrix(630,  2)
1137            NOR.SetSubMatrix(0, 24,  0,  2,  nullMat)
1138            NOR.SetSubMatrix(24, 606, 0,  2,  phi)
1139            H = CST.Append(NOR)                # H = [CST NOR]
1140
1141            # Building  M_full:
1142            M_full  = ModRed.readFedemMatrix(self. _resourcesFolder  + "\\beam.nas" +
        "\\Fedem" + "\\unsrtENodes" + "\\M_full.res")
1143            dict  = ModRed.readMappingFileFedem(self._resourcesFolder + "\\beam.nas" +
        "\\Fedem" + "\\unsrtENodes" + "\\MEQN.res")
1144            eNodes = [22,  28,  6,  44]
```

```
1145            iNodes = [i for i in range(1, 106)]
1146            for e in eNodes:
1147                iNodes.remove(e)
1148            M_full = ModRed.partitionMatrix(M_full, eNodes, iNodes, dict, 6)
1149
1150            gravVec = ModRed.calculateGravityVector(M_full, H, 4, 101, 6)
1151            print(gravVec)
1152
1153            diffs = []
1154            for row in range(0, gravVec.RowCount):
1155                for col in range(0, gravVec.ColumnCount):
1156                    diff = correct[row, col] − gravVec[row, col]
1157                    diffs.append(diff)
1158            print(diffs)
1159            print("max: " + str(max(diffs)))
1160            for row in range(0, gravVec.RowCount):
1161                for col in range(0, gravVec.ColumnCount):
1162                    self.assertAlmostEqual(gravVec[row, col], correct[row, col], places=5)
1163
1164
1165
1166        def test_calculateGravityVector_Ansys_beam(self):
1167            with open(self._resourcesFolder + "\\beam.nas" + "\\Fedem" + "\\gravVec.p") as
          file:
1168                correct = pickle.load(file)
1169
1170            CST = ModRed.readMMFMatrix(self._resourcesFolder + "\\beam.nas" + "\\Ansys" +
          "\\CST.mmf", "full")
1171            NOR = ModRed.readMMFMatrix(self._resourcesFolder + "\\beam.nas" + "\\Ansys" +
          "\\NOR.mmf", "full")
1172            H = CST.Append(NOR)          # H = [CST NOR]
1173
1174            # Building M_full:
1175            M_full = ModRed.readMMFMatrix(self._resourcesFolder + "\\beam.nas" + "\\Ansys"
          + "\\M_full_lumped.mmf", "full")
1176            dict = ModRed.readMappingFile(self._resourcesFolder + "\\beam.nas" + "\\Ansys"
          + "\\M_full_lumped.mapping")
1177            eNodes = [1, 2, 3, 4]
1178            iNodes = [i for i in range(5, 106)]
1179            M_full = ModRed.partitionMatrix(M_full, eNodes, iNodes, dict, 6)
1180
1181            gravVec = ModRed.calculateGravityVector(M_full, H, 4, 101, 6)
1182
1183            diffs = []
1184            for row in range(0, gravVec.RowCount):
1185                for col in range(0, gravVec.ColumnCount):
1186                    diff = correct[row, col] − gravVec[row, col]
1187                    diffs.append(diff)
1188            print(diffs)
1189            print("max: " + str(max(diffs)))
```

```
1190            for row in range(0, gravVec.RowCount):
1191                for col in range(0, gravVec.ColumnCount):
1192                    self.assertAlmostEqual(gravVec[row, col], correct[row, col], places=1)
1193
1194
1195
1196        def test_readMappingFile(self):
1197
1198            path = self._resourcesFolder + "\\twoQUAD4" + "\\Ansys" + "\\M_full.mapping"
1199            dict = ModRed.readMappingFile(path)    # dict[dof] = equation
1200
1201            self.assertEqual(dict[1], 13)
1202            self.assertEqual(dict[2], 14)
1203            self.assertEqual(dict[7], 19)
1204            self.assertEqual(dict[13], 31)
1205            self.assertEqual(dict[14], 32)
1206            self.assertEqual(dict[18], 36)
1207
1208
1209
1210    if __name__ == "__main__":
1211        unittest.main()
```

Adrian Opheim

Enhancing the Analysis Possibilities of Fedem by Performing Model Reduction in ANSYS