Structural Dynamics Assignment

Adrian Opheim

July 29, 2017

Contents

1 About the report

This is the obligatory assignment in the course Structural Dynamics (MW2136) at TUM, Department of Mechanical Engineering. A hangar structure is to be analyzed by a Finite Element (FEM) code in order to find its eigenmodes and transient time response. The geometry of the hangar structure can be seen in the Appendix, section A. The report is split into two parts. The first part, Section 2 and 3, discusses the theory behind the computations, namely the free vibration analysis and Newmark integration for computing the transient time response. The second part, Section 4, presents results from the computations done in MATLAB and discusses issues on verification of the FEM model, accuracy of the free vibration analysis and computational time of time integration. Due to the limited scope of the report, some derivations are skipped. These can be found in the lecture notes for the Structural Dynamics course [1].

2 Free vibration analysis

Free vibration analysis consider the case where no external force are applied to a system. By exploiting the mass and stiffness matrices for the structure, $\boldsymbol{M}$ and $\boldsymbol{K}$, one can find the structure's natural frequencies it will vibrate with. With the $\boldsymbol{M}$ and $\boldsymbol{K}$ matrices constructed, free vibration can be described by the set of equations satisfying

$$\boldsymbol{M}\ddot{\boldsymbol{q}} + \boldsymbol{K}\boldsymbol{q} = \boldsymbol{0} \tag{2.1}$$

where $\boldsymbol{M}, \boldsymbol{K}$ are the mass and stiffness matrix respectively, and $\boldsymbol{q}, \ddot{\boldsymbol{q}}$ are the set of degrees of freedom describing the system corresponding to eq. (4.2).
As we know that free vibration motions can be described by a harmonic time response, the solution of eq. (2.1) must be on the form

$$\boldsymbol{q}(t) = \boldsymbol{x}(\alpha\cos(\omega t) + \beta\sin(\omega t)) \tag{2.2}$$

Substituting eq. (2.2) into eq. (2.1), the free vibration of the system is described by

$$(\boldsymbol{K} - \omega^2\boldsymbol{M})\boldsymbol{x} = \boldsymbol{0} \tag{2.3}$$

where ω is the circular frequency, in $\frac{\text{rad}}{\text{s}}$, of the mode shape $\boldsymbol{x}$.

This set of n equations satisifies a solution $\boldsymbol{x}_{(r)}$ such that

$$(\boldsymbol{K} - \omega_r^2\boldsymbol{M})\boldsymbol{x}_{(r)} = \boldsymbol{0} \tag{2.4}$$

Eq. (2.4) tells us that there exist free vibration motions that are described by a so-called eigenmode $\boldsymbol{x}_{(r)}$ and an associated eigenfrequency ω_r such that inertia forces exactly compensate the elastic forces.

2.1 The inverse iteration implementation

In order to find the associated eigenmodes and -frequencies to the eigenvalue problem

$$\boldsymbol{K}\boldsymbol{x} = \omega^2\boldsymbol{M}\boldsymbol{x} \tag{2.5}$$

the *inverse iteration* technique is shown effective. It is a refinement of the *power iteration method* that computes the first associated eigenmode of eq. (2.5) by iteration of the equations

$$\begin{aligned} \boldsymbol{z}_{p+1}^{*} &= \boldsymbol{D}\boldsymbol{z}_{p} \\ \boldsymbol{z}_{p+1} &= \frac{\boldsymbol{z}_{p+1}^{*}}{\|\boldsymbol{z}_{p+1}^{*}\|} \end{aligned} \tag{2.6}$$

where $\boldsymbol{D} = \boldsymbol{K}^{-1}\boldsymbol{M}$ is the so-called dynamic flexibility matrix. $\boldsymbol{z}_{p+1}$ will after a sufficently number of iterations tend to

$$\begin{aligned} \boldsymbol{z}_{p} &\to \lambda_1^{p}\alpha_1\boldsymbol{x}_{(1)} \\ \boldsymbol{z}_{p+1} &\to \lambda_1^{p+1}\alpha_1\boldsymbol{x}_{(1)} \end{aligned} \tag{2.7}$$

Its convergence progress is illustrated in Figure (1).

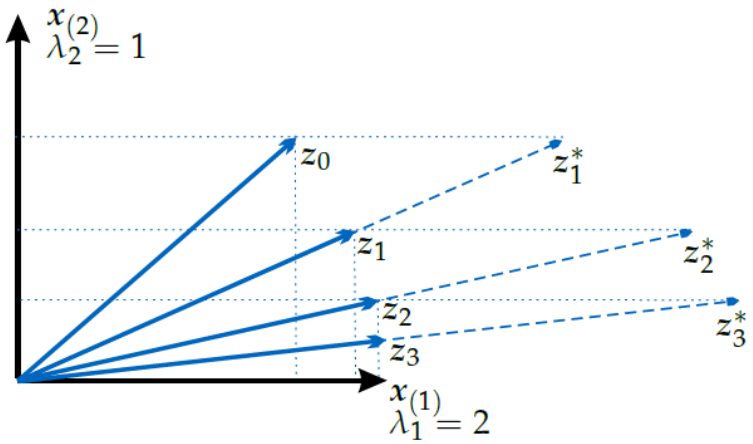

Figure 1: Convergence of the power iteration method with $\lambda_1 = 2$ and $\lambda_2 = 1$. Figure taken from lecture notes in Structural Dynamics [1]

By rewriting the power iteration, it can be expressed without using the computationally expensive $\boldsymbol{D}$ matrix. Starting from eq. (2.6), it can be rewritten as

$$\begin{aligned} \boldsymbol{z}_{p+1} &= \boldsymbol{D}\boldsymbol{z}_{p} \\ &= (\boldsymbol{K}^{-1}\boldsymbol{M})\boldsymbol{z}_{p} \\ \boldsymbol{K}\boldsymbol{z}_{p+1} &= \boldsymbol{M}\boldsymbol{z}_{p} \end{aligned}$$

which is the inverse iteration. Rewritten in iterative fashion as

$$\begin{aligned} \boldsymbol{y}_{p} &= \boldsymbol{M}\boldsymbol{z}_{p} \\ \boldsymbol{K}\boldsymbol{z}_{p+1} &= \boldsymbol{y}_{p} \end{aligned} \tag{2.8}$$

one can see its benefits over the power iteration: No computationally expensive flexibility matrix $\boldsymbol{D}$ needs to be calculated, as the last equation in (2.8) is solved by factorizing $\boldsymbol{K}$ and then applying forward and backward substitution.

2.2 Applying orthogonal deflation

By orthogonal deflation, one uses the fact that eigenmodes are orthogonal to each other. It can be shown, [1] p. 59-60, that two eigenmodes of distinct eigenfrequencies, denoted $\boldsymbol{x}_{(r)}$ and $\boldsymbol{x}_{(s)}$, have the orthogonality relationship

$$\boldsymbol{x}_{(s)}^{T}\boldsymbol{M}\boldsymbol{x}_{(r)} = 0 \tag{2.9}$$

and

$$\boldsymbol{x}_{(s)}^T \boldsymbol{K} \boldsymbol{x}_{(r)} = 0 \tag{2.10}$$

telling us that the two eigenmodes $\boldsymbol{x}_{(r)}$ and $\boldsymbol{x}_{(s)}$ exist in separate dimensions which are orthogonal to each other. Knowing this, it is possible to extract the higher eigenmodes. This is the concept of orthogonal deflation. By applying the orthogonal projection operator

$$\boldsymbol{P}_j = \mathbf{I} - \boldsymbol{x}_{(j)} \boldsymbol{x}_{(j)}^T \boldsymbol{M} \qquad \text{with } j = 1, ..., k \tag{2.11}$$

to eigenmode $\boldsymbol{x}_{(k)}$ when k eigenmodes are known, one "removes" the influence of the k earlier eigenmodes, and thus projecting $\boldsymbol{x}_{(k)}$ onto the dimension wanted, $\boldsymbol{x}_{(k+1)}$. The orthogonal projection operator can therefore be rewritten as

$$\boldsymbol{P} = \boldsymbol{P}_1 \cdot \boldsymbol{P}_2 \cdot \ldots \cdot \boldsymbol{P}_k = \mathbf{I} - \boldsymbol{x}_{(1)} \boldsymbol{x}_{(1)}^T \boldsymbol{M} - \boldsymbol{x}_{(2)} \boldsymbol{x}_{(2)}^T \boldsymbol{M} - \ldots - \boldsymbol{x}_{(k)} \boldsymbol{x}_{(k)}^T \boldsymbol{M} \tag{2.12}$$

Thus, by applying the orthogonal projection operator $\boldsymbol{P}$ to the inverse iteration, it is possible to extract higher eigenmodes. The inverse iteration, eq. (2.8) can therefore be expanded as follows at iteration step p when searching for eigenmode $k+1$, knowing all k previous:

$$\begin{aligned} \boldsymbol{z}_p &= \boldsymbol{P} \boldsymbol{z}_0 \\ \boldsymbol{y}_p &= \boldsymbol{M} \boldsymbol{z}_p \\ \boldsymbol{K} \hat{\boldsymbol{z}}_{p+1} &= \boldsymbol{y}_p \\ \boldsymbol{z}_{p+1}^* &= \boldsymbol{P} \hat{\boldsymbol{z}}_{p+1} \\ \boldsymbol{z}_{p+1} &= \frac{\boldsymbol{z}_{p+1}^*}{\|\boldsymbol{z}_{p+1}^*\|} \end{aligned} \tag{2.13}$$

After a sufficient number of iterations, $\boldsymbol{z}_{p+1}$ will converge to the desired eigenmode $\boldsymbol{z}_{k+1}$.

3 Time integration with the Newmark method

The Newmark method, developed by N.M. Newmark, is a single-step integration formula, commonly used to solve the dynamic equation

$$\boldsymbol{M}\ddot{q} + \boldsymbol{C}\dot{\boldsymbol{q}} + \boldsymbol{K}\boldsymbol{q} = \boldsymbol{p}(t) \tag{3.1}$$

The method uses a Taylor series expansion of the displacements and velocities to find the state vector of the system at time $t_{n+1} = t_n + h$. The approximation formulas for the Newmark method is given as [1]

$$\begin{aligned} \dot{\boldsymbol{q}}_{n+1} &= \dot{\boldsymbol{q}}_n + (1-\gamma) h \ddot{\boldsymbol{q}}_n + \gamma h \ddot{\boldsymbol{q}}_{n+1} \\ \boldsymbol{q}_{n+1} &= \boldsymbol{q}_n + h\dot{\boldsymbol{q}}_n + h^2(\frac{1}{2} - \beta)\ddot{\boldsymbol{q}}_n + h^2 \beta \ddot{\boldsymbol{q}}_{n+1} \end{aligned} \tag{3.2}$$

As these formulas depend on the acceleration at time t_{n+1}, namely $\ddot{\boldsymbol{q}}_{n+1}$, one can insert the Newmark approximation formulas (3.2) into the equations of motion at time t_{n+1} in order to compute the acceleration $\ddot{\boldsymbol{q}}_{n+1}$:

$$\big[\boldsymbol{M} + \gamma h \boldsymbol{C} + \beta h^2 \boldsymbol{K}\big] \ddot{\boldsymbol{q}}_{n+1} = \boldsymbol{p}_{n+1} - \boldsymbol{C}\big[\dot{\boldsymbol{q}}_n + (1-\gamma) h \ddot{\boldsymbol{q}}_n\big] - \boldsymbol{K}\big[\boldsymbol{q}_n + h\dot{\boldsymbol{q}}_n + (\frac{1}{2} - \beta) h^2 \ddot{\boldsymbol{q}}_n\big] \tag{3.3}$$

This step implies solving a linear set of equations associated with the so-called time-stepping matrix

$$\boldsymbol{S} = \big[\boldsymbol{M} + \gamma h \boldsymbol{C} + \beta h^2 \boldsymbol{K}\big] \tag{3.4}$$

which can be seen as an inertia matrix, as it relates accelerations at t_{n+1} to a pseudo-force. The Newmark integration first computes $\ddot{\boldsymbol{q}}_{n+1}$ from (3.3) which further on is used in (3.2) to compute displacements and velocities. The Newmark integration scheme can be seen in the Appendix, Figure 8

[1] The derivation of the Newmark equations can be seen at [1], p. 119-120.

3.1 Stability of the Newmark scheme

By doing a stability analysis of the Newmark algorithm, one can show that the integration scheme is stable if

$$\gamma \geq \frac{1}{2} \tag{3.5}$$

and

$$(\gamma + \frac{1}{2})^2 - 4\beta \leq \frac{4}{\omega^2 h^2} \tag{3.6}$$

From this one can see that if $\beta \geq \frac{1}{4}(\gamma + \frac{1}{2})^2$, the step size h can be chosen freely - it is unconditionally stable. The stability regions for the Newmark scheme are shown in the Appendix, Figure 9.

3.2 Implicit Newmark scheme

The Newmark equations are given implicit as (3.2) when the constants γ and β are chosen so that $\ddot{\boldsymbol{q}}_{n+1}$ needs to be determined by eq. (3.3). This is true for the case when one assumes constant average acceleration. By this, you assume constant acceleration between time-steps t_n and t_{n+1}, namely that

$$\dot{\boldsymbol{q}}(\tau) = \frac{\ddot{\boldsymbol{q}}_n + \ddot{\boldsymbol{q}}_{n+1}}{2} \tag{3.7}$$

which corresponds to the case where $\gamma = \frac{1}{2}$ and $\beta = \frac{1}{4}$.
From the stability analysis, it is evident that $\beta \geq \frac{1}{4}(\gamma + \frac{1}{2})^2$, and the average constant acceleration assumption is thus unconditionally stable.

3.3 Explicit Newmark scheme

The purely explicit integration scheme of Newmark is seen when $\gamma = \beta = 0$. This is though not used, because the integration scheme will always be unstable, due to the fact that eq. (3.6) is never fulfilled. As the purely explicit integration scheme is only dependent on the previous time-steps t_n, one can see it as a Euler forward formula, $\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + h\dot{\boldsymbol{u}}_n$.

When using an explicit integration scheme, it must then be done "semi-explicit", namely with the central difference algorithm. Here, $\gamma = \frac{1}{2}, \beta = 0$. The approximation formulas for Newmark, eq. (3.2) then writes

$$\begin{aligned} \dot{\boldsymbol{q}}_{n+1} &= \dot{\boldsymbol{q}}_n + \frac{h}{2}(\ddot{\boldsymbol{q}}_n + \ddot{\boldsymbol{q}}_{n+1}) \\ \boldsymbol{q}_{n+1} &= \boldsymbol{q}_n + h\dot{\boldsymbol{q}}_n + \frac{h^2}{2}\ddot{\boldsymbol{q}}_n \end{aligned} \tag{3.8}$$

These equations become "semi-explicit" by the fact that $\boldsymbol{q}_{n+1}$ can be computed directly, but the acceleration needed for computing $\dot{\boldsymbol{q}}_{n+1}$ must be deduced from

$$\boldsymbol{M}\ddot{\boldsymbol{q}}_{n+1} = \boldsymbol{p}_{n+1} - \boldsymbol{K}\boldsymbol{q}_{n+1} \tag{3.9}$$

For the central difference algorithm to be stable, the stability limit eq. (3.6) gives $\omega h \leq 2$ where ω is chosen as the highest frequency contained in the model. This means that the time-steps h required for the central difference algorithm to be stable must be very small. It is therefore normally used for high-frequency cases like computing impact response.

4 Results

4.1 Verifying the Finite Element Model

The given FEM model and the corresponding assembled stiffness matrices $\boldsymbol{K}$ and $\boldsymbol{M}$ need to be verified. Before fixing any of the degrees of freedom, one can verify that the elastic forces created in a unit translation equal to zero, namely

$$\boldsymbol{K}\boldsymbol{u}_{\boldsymbol{trans}} = 0 \tag{4.1}$$

where $\boldsymbol{u}_{\boldsymbol{trans}}$ is a translation rigid body mode with unit amplitude.
The structure consists of 76 elements, resulting in 44 active nodes and by that $44 \times 6 = 264$ degrees of freedom before fixing any nodes. As every element has its degrees of freedom expressed as

$$\boldsymbol{q}_{\boldsymbol{eL}}^T = \begin{bmatrix} u_1 & v_1 & w_1 & \psi_{x_1} & \psi_{y_1} & \psi_{z_1} & u_2 & v_2 & w_2 & \psi_{x_2} & \psi_{y_2} & \psi_{z_2} \end{bmatrix} \tag{4.2}$$

one could set $u_1 = u_2 = 1$ to obtain a unit translation in the local (x, y, z) coordinate system, corresponding to Figure 2.

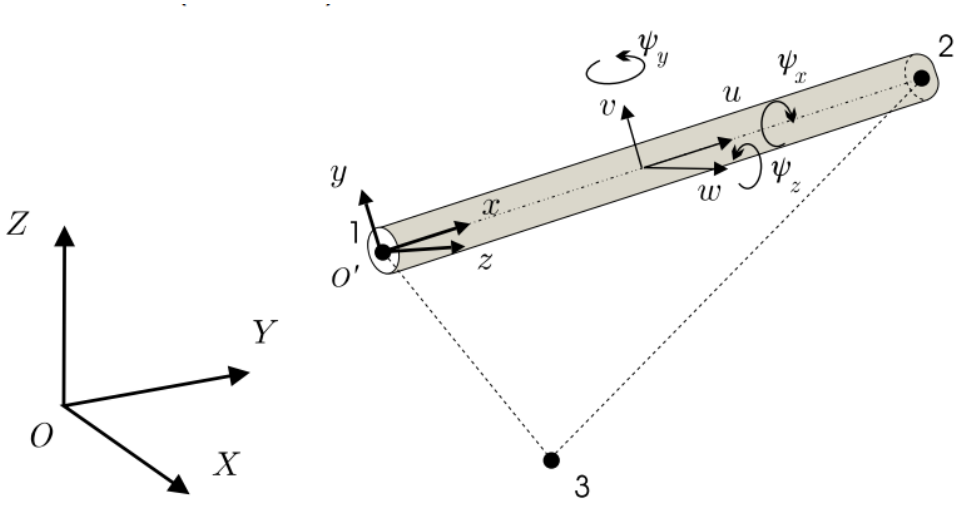

Figure 2: Degrees of freedom in a three-dimensional beam element. Figure taken from lecture notes in Structural Dynamics [1]

The results from having set $u_1 = u_2 = 1, v_1 = v_2 = 1$ and $w_1 = w_2 = 1$ and performing eq. (4.1) in MATLAB [2] are summarized in Table 1.

	$u_1 = u_2 = 1$	$v_1 = v_2 = 1$	$w_1 = w_2 = 1$
$\boldsymbol{K}\boldsymbol{u}_{trans}$	7.4506×10^{-9}	2.7649×10^{-10}	1.0987×10^{-8}

Table 1: Maximum entries in `null_check` vector after setting a unit translational displacement.

Due to the round-off errors from numerical calculations, all entries are not zero, but close. The maximum aquired entry is at $w_1 = w_2 = 1$, and is 1.0987×10^{-8}, making it reasonable to conclude that the assembled $\boldsymbol{K}$ matrix is correct.

In order to verify the assembled mass matrix $\boldsymbol{M}$, it is possible to compute the total mass of the structure by

[2] The resulting vector is in the MATLAB script called `null_check`

$$\boldsymbol{u}_{\boldsymbol{trans}}^T \boldsymbol{M} \boldsymbol{u}_{\boldsymbol{trans}} = m_{total} \tag{4.3}$$

Applying this, m_{total} is calculated to be 1.5949×10^4kg. [3]

In order to verify this result, one can find the mass of every element by multiplying its density ρ with its area and length. Doing this, the resulting, calculated mass is 1.6053×10^4kg, [4] differing only 0.65% from the m_{total} found by (4.3). This is also considered to be a reasonable result, and the constructed $\boldsymbol{M}$ matrix seems correct.

4.2 Results from free vibration analysis

For the free vibration analysis, the first 10 eigenmodes of the hangar structure are found using the technique of inverse iteration with orthogonal deflation, following the scheme of eq. (2.8). $\boldsymbol{z}_0$, which is used in every iteration together with the projection operator, is set to a $(n \times 1)$ vector containing only ones. In order to solve $\boldsymbol{K}\hat{\boldsymbol{z}}_{p+1} = \boldsymbol{y}_p$, $\boldsymbol{K}$ is LU factorised and then solved for $\hat{\boldsymbol{z}}_{p+1}$ by forward- and backwards-substitution. In order to determine the convergence of the method, two successive Rayleigh quotients are found from

$$\omega^2_{\text{estimate},p+1} = \frac{\boldsymbol{z}_{p+1}^T \boldsymbol{K} z_{p+1}}{\boldsymbol{z}_{p+1}^T \boldsymbol{M} z_{p+1}} \tag{4.4}$$

From this, one could say that the method has converged if the difference between to successive Rayleigh quotient is sufficiently small:

$$\left|\omega^2_{\text{estimate},p+1} - \omega^2_{\text{estimate},p}\right| < \varepsilon \tag{4.5}$$

The inverse iteration implementation in MATLAB can be seen in the Appendix, Section F.1. When choosing $\varepsilon = 10^{-4}$, the implementation needed in average 20.3 iterations to converge to the first 10 eigenmodes, with a maximum of 61 iterations (eigenmode 6), and minimum of 8 iterations (eigenmode 2). Eigenmodes found from inverse iteration can be seen in Table 2

i	ω_i [Hz]
1	0.2272
2	0.2628
3	0.3832
4	0.4651
5	0.5459
6	0.5952
7	0.6472
8	0.7266
9	0.8083
10	1.1245

Table 2: The first 10 eigenfrequencies obtained from the inverse iteration

The found eigenfrequencies from the inverse iteration are compared to the eigenfrequencies found from MATLAB's own Eigenmode solver, `eig`, which solve the system

$$\boldsymbol{AV} = \boldsymbol{BVD} \tag{4.6}$$

[3] Corresponds to the variable `m_total`

[4] Corresponds to the variable `m_total_calc`

with the command `[V, D] = eig(A, B)`, corresponding to the free vibration problem $\boldsymbol{K}\boldsymbol{x} = \omega^2\boldsymbol{M}\boldsymbol{x}$. The difference, in percent, between the found eigenfrequencies from inverse iteration and the `eig` function are calculated. [5] The most significant difference is found for the first eigenmode, with a difference of 0.0022%.

4.3 Results from time integration of transient dynamics

4.3.1 The implicit case

The average constant acceleration algorithm has been implemented in MATLAB, following the Newmark integration scheme, which can be seen in the Appendix, section D.1. The `newmark` function calculates the displacements, velocities and accelerations independent of which Newmark algorithm is used. For the implicit case, three different step sizes are used to illustrate the method's stability, independent of the step size.

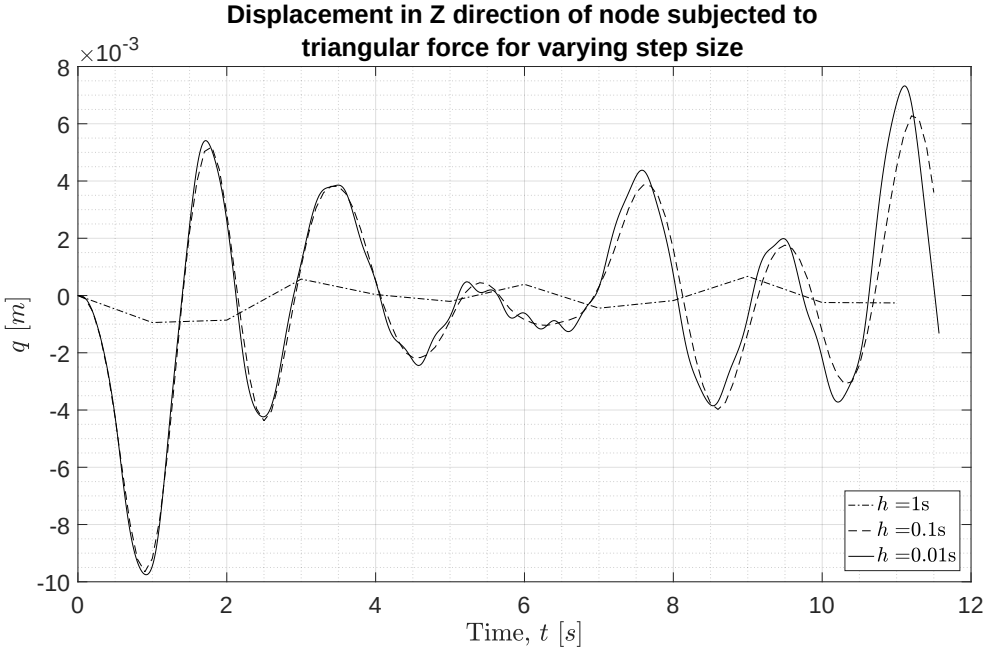

Figure 3: Displacement of node 51, i.e. the node being subjected to a triangular force $p(t)$, in global Z direction

From Figure 3 one can see how the solution is gradually increasing in accuracy as the step size h gets lower. The periodicity error for the implicit case is also calculated. It is defined as $(T_{num} - T)/T$, where T is the period of the free oscillation and T_{num} is the period of the compute response. For the implicit case, this is found as $\frac{\omega^2 h^2}{12}$. The periodicity errors for every step size are summarized in Table 3.

4.3.2 The explicit case

For the explicit case of Newmark integration, the central difference algorithm is implemented. In order to find the conditional stability limit for this scheme, $\omega h \leq 2$, ω must be chosen as the highest eigenfrequency contained in the model. This is done by taking the highest found frequency from the `eig` solver in MATLAB. For the hangar structure, this is 366.36Hz, giving a stability limit step size of $h \leq 8.957 \times 10^{-4}$s. The stability limit is illustrated in Figure 4. There one can see how the solution "explodes" and becomes unstable for a step size just 0.01% higher than the stability limit.

[5] Corresponds to the `omega_accuracy` vector

h	Periodicity error
1s	0.0180
0.1s	1.8×10^{-4}
0.01s	1.8×10^{-6}

Table 3: Periodicity error of the implicit Newmark integration scheme with the average constant acceleration implementation.

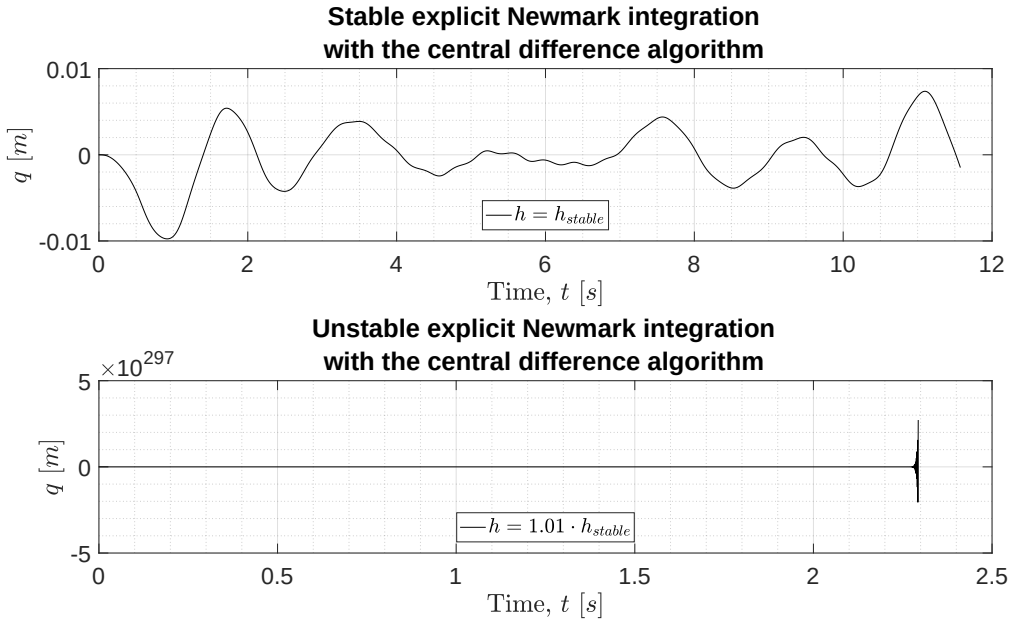

Figure 4: The explicit Newmark scheme with step size respectively within and outside the stability limit

4.3.3 Computational costs for the explicit and implicit case

Table 4 shows how the computational time for the implicit and explicit time integration varies. The lowest step size, $h = 8.96 \times 10^{-4}$, is set equal to the highest stable step size, h_{stable}. The results show, surprisingly, that the implicit time Newmark integration is marginally faster than the explicit when the step size gets smaller. This is due to the implementation of the `Newmark` function in MATLAB. Both of the implementations are run through the same algorithm, which is not advantegous for the explicit case. An implementation specific for the explicit case, following equations (3.8) and (3.9) would reduce the computational time of the explicit case.

	Step size h	Number of time steps	Computational time
	8.96×10^{-4}s	12923	9.99s
Implicit Newmark integration	5×10^{-4}s	23150	17.47s
	1×10^{-4}s	115752	90.04s
	8.96×10^{-4}s	12923	11.52s
Explicit Newmark integration	5×10^{-4}s	23150	21.79s
	1×10^{-4}s	115752	92.99s

Table 4: Computational time for the explicit and implicit Newmark integration

5 References

[1] Daniel J. Rixen. *Structural Dynamics*. Fachschaft Maschinenbau, Version 2017.0.

Appendices

A The hangar structure

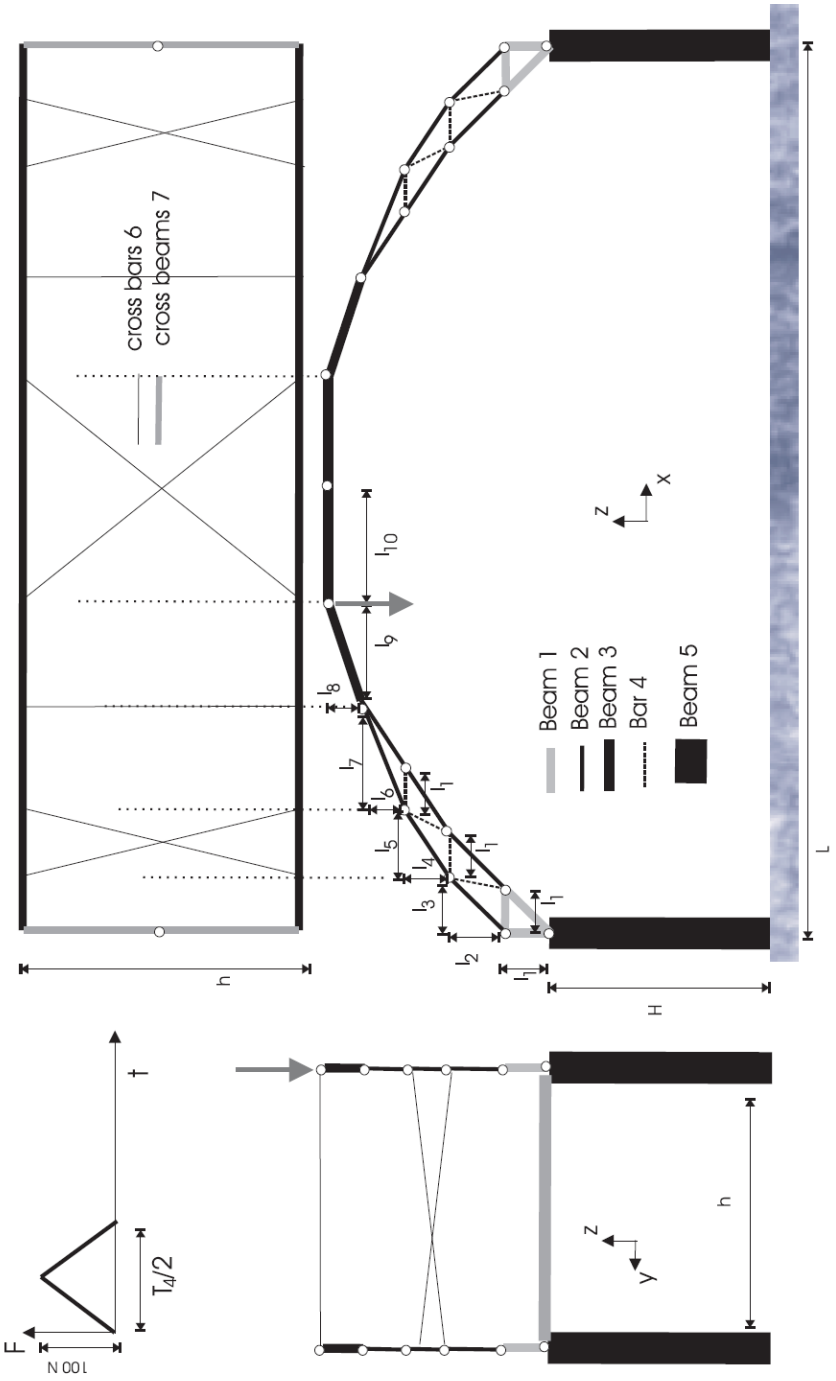

Figure 5: Geometry of the hangar structure and the applied triangular force F.

B First 10 eigenmodes of the hangar structure

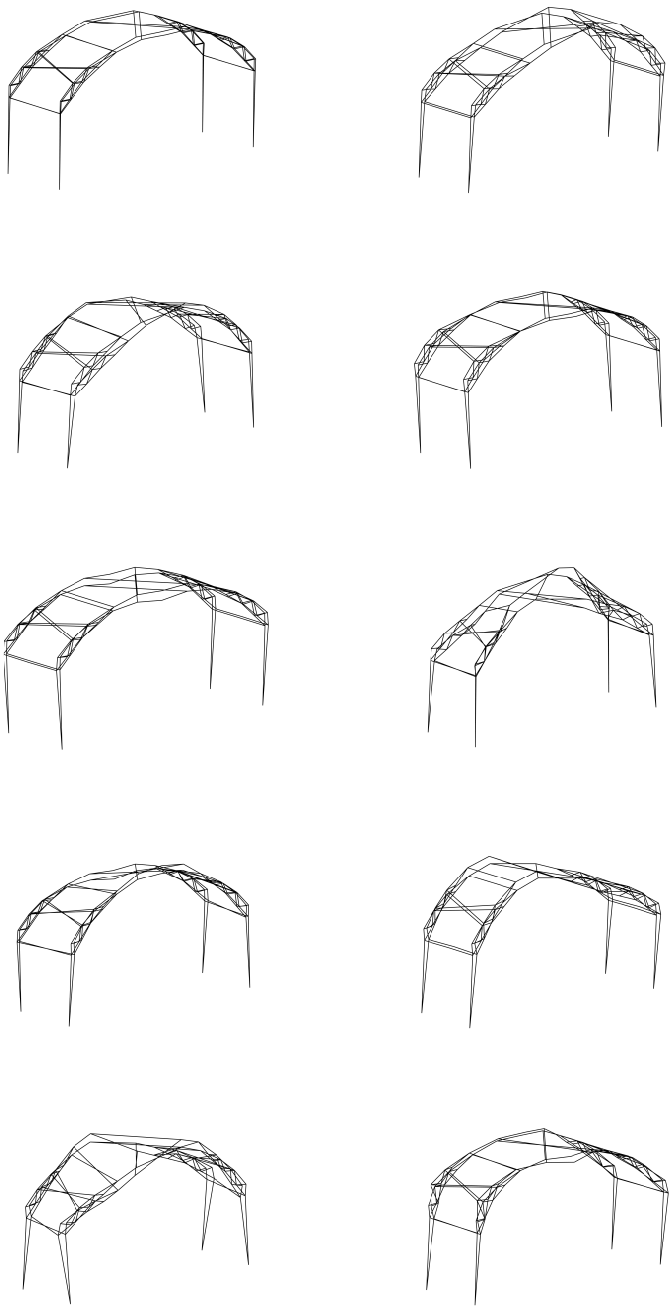

Figure 6: Resulting displacement from the first 10 eigenmodes

C Displacement from applied force

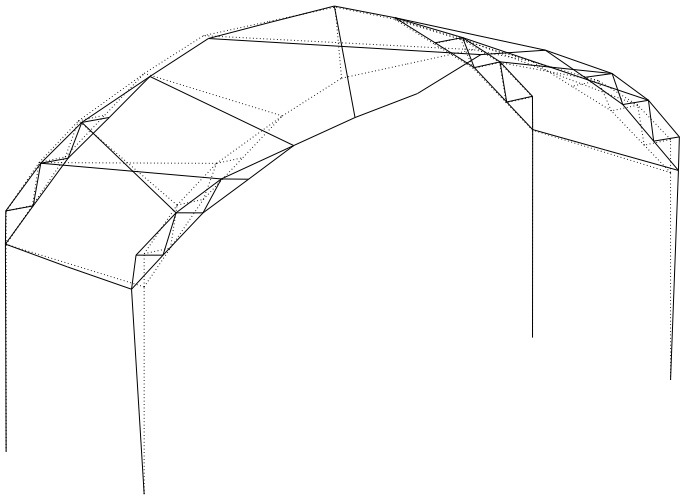

Figure 7: Resulting displacement from the applied triangular force. Displacement taken 1s after force is applied. Dotted line shows original geometry of the structure. Black line is the resulting displacement, amplified with a factor of 500.

D Newmark integration figures

D.1 Newmark integration scheme

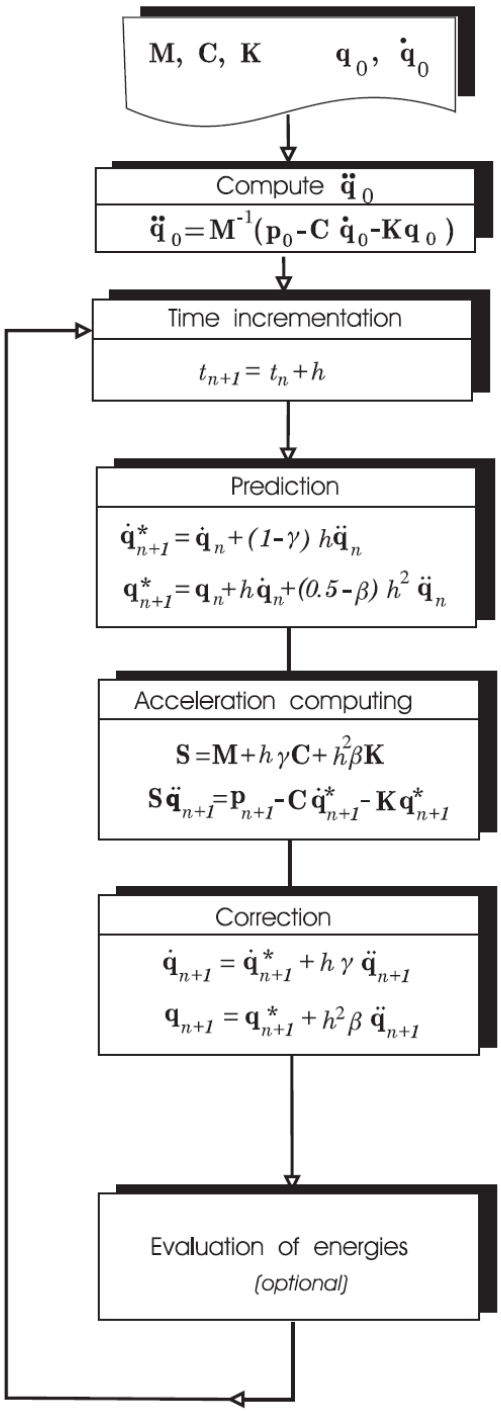

Figure 8: Newmark integration scheme for linear systems. Figure taken from lecture notes in Structural Dynamics [1], p. 121

D.1.1 Stability regions for the Newmark algorithm

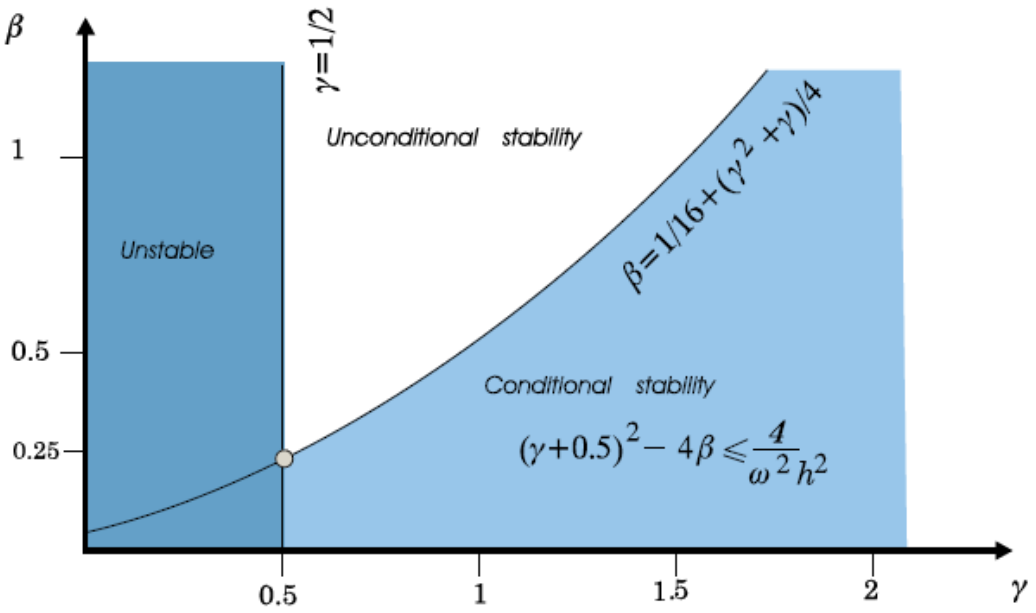

Figure 9: Stability regions for the Newmark algorithm. Figure taken from lecture notes in Structural Dynamics [1], p. 123

D.1.2 Overview of all Newmark integration schemes

Algorithm	γ	β	Stability limit ωh	Amplitude error $\rho - 1$	Periodicity error $\Delta T / T$
Purely explicit	0	0	0	$\frac{\omega^2 h^2}{4}$	—
Central difference	$\frac{1}{2}$	0	2	0	$\frac{-\omega^2 h^2}{24}$
Fox & Goodwin	$\frac{1}{2}$	$\frac{1}{12}$	2.45	0	$O(h^3)$
Linear acceleration	$\frac{1}{2}$	$\frac{1}{6}$	3.46	0	$\frac{\omega^2 h^2}{24}$
Average constant acceleration	$\frac{1}{2}$	$\frac{1}{4}$	∞	0	$\frac{\omega^2 h^2}{12}$
Average constant acceleration (modified)	$\frac{1}{2}+\alpha$	$\frac{(1+\alpha)^2}{4}$	∞	$-\alpha\frac{\omega^2 h^2}{2}$	$\frac{\omega^2 h^2}{12}$

Figure 10: All integration schemes for the Newmark integration with the corresponding stability limits and error measurements. Figure taken from lecture notes in Structural Dynamics [1], p. 124

E Implicit integration plots

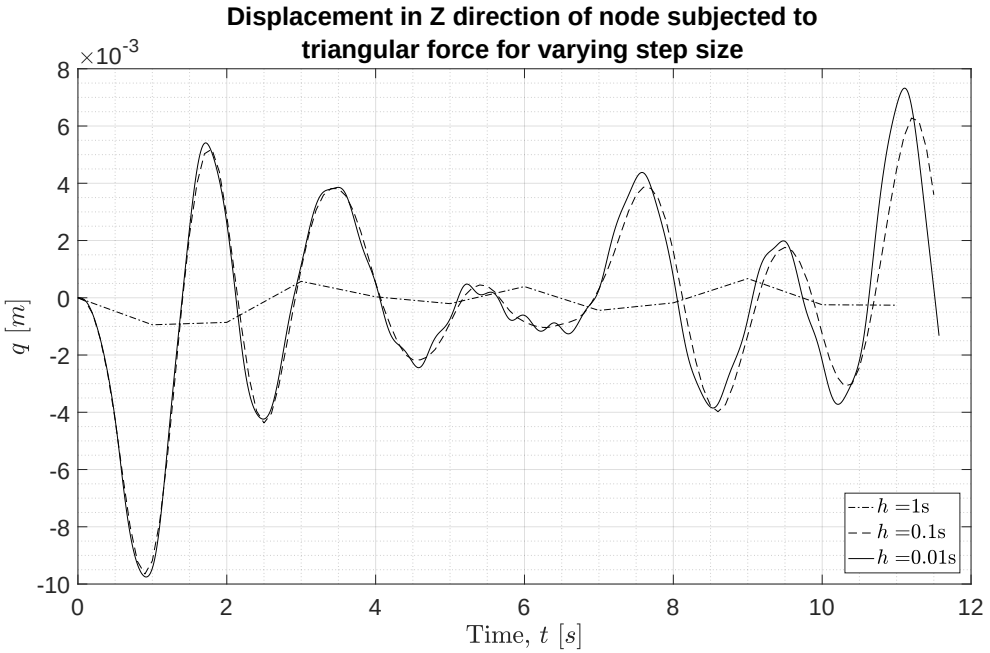

Figure 11: Displacement of node 45, i.e. the node being subjected to a triangular force $p(t)$, in global Z direction

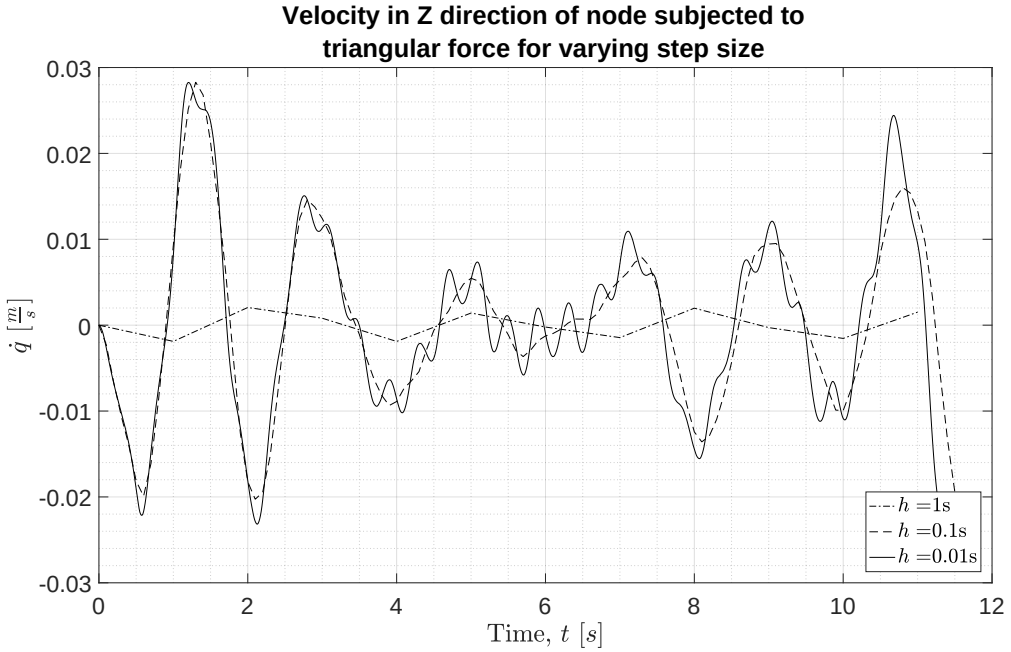

Figure 12: Velocity of node 45, i.e. the node being subjected to a triangular force $p(t)$, in global Z direction

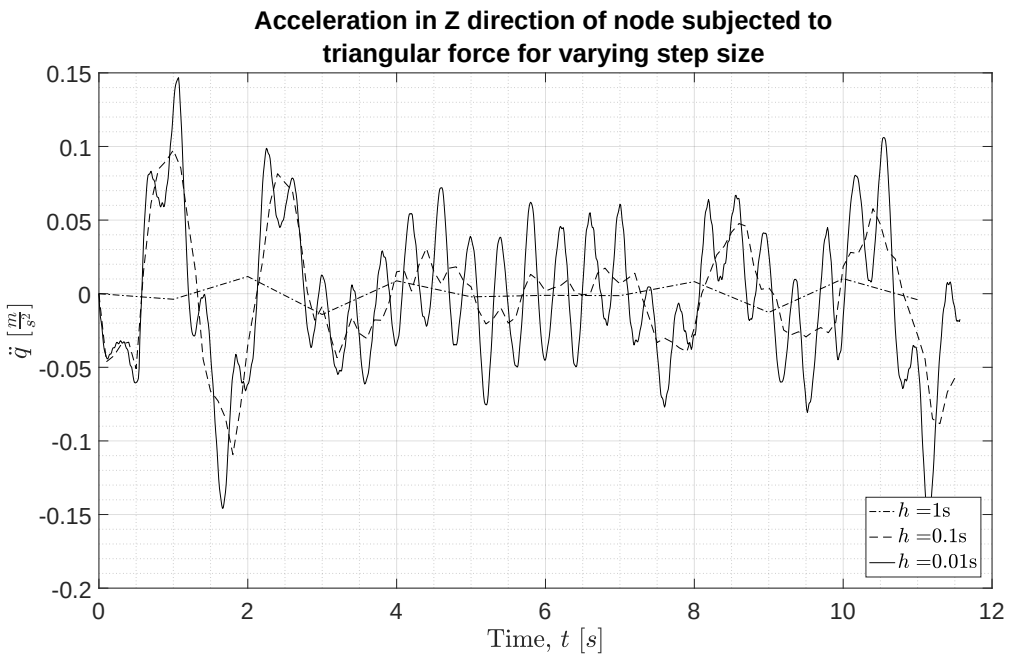

Figure 13: Acceleration of node 45, i.e. the node being subjected to a triangular force $p(t)$, in global Z direction

F MATLAB code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Finite Element assembly of a beam-bar 3D structure
%
%    Project part of the course Structural Dynamics, TUM MW 2136
%    D.R. 05.06.2017
%
%   This file contains the assembly of a general 3D mesh consisting of beam
%   and bar elements. The nodal, material and connectivity matrices are
%   defined in the 'Nodes', 'Mat' and 'Elements' matrices repsectively. The
%   example given here is a hanguar as described in a joined document.
%   The beam element matrices are given. The building of the K and M matrices, and
%   definining the geometry of the structure is done by Daniel Rixen.
%   Implementation of the inverse iteration and Newmark integration done by
%   Adrian Opheim, July 2017.
%
%%%%%%%%%%%%%%%%%%%%%%

 clear
 close all

%%%%%%%%%% Geometric parameters %%%%%%%%%%%%%%
%
% define here all the geometric quantities you need (diameters, lengths, thinkness, etc.)

L=80;
l1=4;
l2=5;
l3=5;
l4=4;
l5=6;
l6=4;
l7=10;
l8=3;
l9=9;
l10=10;

h=25;

H=25;

D1=0.20; d1=0.195;    % beams group 1 (lower arch)
D2=0.20; d2=0.195;    % beams group 2 (mid arch)
D3=0.25; d3=0.244;    % beams group 3 (high arch and span)
D4=0.05; d4=0.046;    % bars in arch
D5=0.40; d5=0.380;    % column beams
D6=0.08; d6=0.075;    % cross bars
D7=0.25; d7=0.24;    % cross beams


%  Material properties
%
% define here the material constant.

 E=2.1e11;
 nu=0.3;
 rho=7500;

%%%%%%% Computing characteristics %%%%%%%%%%%%%%
%
% Here you compute the quantities needed for the 'Mat' array (see below).
%

 G = E/(2*(1+nu));

 A1  = pi*(D1^2-d1^2)/4;
 A2  = pi*(D2^2-d2^2)/4;
 A3  = pi*(D3^2-d3^2)/4;
```

```
 A4  = pi*(D4^2-d4^2)/4;
 A5  = pi*(D5^2-d5^2)/4;
 A6  = pi*(D6^2-d6^2)/4;
 A7  = pi*(D7^2-d7^2)/4;

 I1  = pi*(D1^4-d1^4)/64; J1  = pi*(D1^4-d1^4)/32;
 I2  = pi*(D2^4-d2^4)/64; J2  = pi*(D2^4-d2^4)/32;
 I3  = pi*(D3^4-d3^4)/64; J3  = pi*(D3^4-d3^4)/32;
 I5  = pi*(D5^4-d5^4)/64; J5  = pi*(D5^4-d5^4)/32;
 I7  = pi*(D7^4-d7^4)/64; J7  = pi*(D7^4-d7^4)/32;

%%%%%%%  Nodes    %%%%%%%%
%  x    ,   y   ,   z
%
% Create here the 'Nodes' matrix. It contains the nodal coordinates x,y,z.
%

Nodes(1,:)  = [ 0               0       H ];    % arch
Nodes(2,:)  = [ 0               0       H+l1];
Nodes(3,:)  = [ l1              0       H+l1];
Nodes(4,:)  = [ l3              0       H+l1+l2];
Nodes(5,:)  = [ l3+l1           0       H+l1+l2];
Nodes(6,:)  = [ l3+l5           0       H+l1+l2+l4];
Nodes(7,:)  = [ l3+l5+l1        0       H+l1+l2+l4];
Nodes(8,:)  = [ l3+l5+l7        0       H+l1+l2+l4+l6];
Nodes(9,:)  = [ l3+l5+l7+l9     0       H+l1+l2+l4+l6+l8];   %This is the node where the force is
applied.
Nodes(10,:) = [ l3+l5+l7+l9+l10 0       H+l1+l2+l4+l6+l8]; % This is the middle node of the arch
for i=21:29,
    Nodes(i,:) = Nodes(i-20,:);
    Nodes(i,1) = L-Nodes(i,1);
end
Nodes(30,:)   = [0     0    0]; % ground nodes
Nodes(31,:)   = [L     0    0];
for i=1:31,
   Nodes(i+100,:) = Nodes(i,:) + [0 h 0];
end
Nodes(135,:) = [0 h/2 H];
Nodes(136,:) = [L h/2 H];

%%%%%%% Dummy node for beams %%%%%%%%%
%
% this node is use to orient the bam cross section in 3D (see lecture
% notes).
% Since in this example the cross section is axisymmetric any orientation
% is fine. This we choose a dummy node randonly, just making sure it is not
% colinear with the 2 nodes of a beam.

 Nodes(200,:) = [0 40 0];

%%%%%%%  Material and section properties    %%%%%
%
%  E    ,   G   , rho ,   A   ,  Ix  ,   Iy   , Iz
%
% Matrix 'Mat' contains the material and section properties definition. Here below an example with
% 7 different materials. Materials 1, 2, 3, 5 and 7 are for beam elements, materials 4 and 6 are for
bar elements.

 Mat(1,1:7)=  [ E   G   rho   A1  J1    I1     I1];  % beams group 1 (lower arch)
 Mat(2,:)   =  [ E   G   rho   A2  J2    I2     I2];  % beams group 2 (mid arch)
 Mat(3,:)   =  [ E   G   rho   A3  J3    I3     I3];  % beams group 3 (high arch and span)
 Mat(4,:)   =  [ E   G   rho   A4  0      0      0];  % bars in arch
 Mat(5,:)   =  [ E   G   rho   A5  J5    I5     I5];  % column beams
 Mat(6,:)   =  [ E   G   rho   A6  0      0      0];  % cross bars
 Mat(7,:)   =  [ E   G   rho   A7  J7    I7     I7];  % cross beams

%%%%%%% Elements   %%%%%%%%%
%
```

```
%  type(1=bar,2=beam), mat, node 1 , node 2, node 3(for beam)
%
%  The Matrix 'Elements' contains the connectivity (localization) informations for all the elements
of the mesh,
%  together with the material and geometric related properties.
%  For example, here below: element no.1 is a beam (type=2), is made of
%  material 7, connects node 1 and node 135 and has the dummy node 200
%  (necessary for a beam)

Elements(1,:)  = [2   1       1  2  200];  % arch
Elements(2,:)  = [2   1       1  3  200];
Elements(3,:)  = [2   1       2  3  200];
Elements(4,:)  = [2   2       2  4  200];
Elements(5,:)  = [2   2       3  5  200];
Elements(6,:)  = [2   2       4  6  200];
Elements(7,:)  = [2   2       5  7  200];
Elements(8,:)  = [2   2       6  8  200];
Elements(9,:)  = [2   2       7  8  200];
Elements(10,:) = [2   3       8  9  200];
Elements(11,:) = [1   4       3  4  200];
Elements(12,:) = [1   4       4  5  200];
Elements(13,:) = [1   4       5  6  200];
Elements(14,:) = [1   4       6  7  200];
Elements(15,:) = [2   3       9  10 200];   % middle span
Elements(16,:) = [2   3       10 29 200];
% create right side of arch
for i=1:14,
    Elements(16+i,:) = Elements(i,:);
    Elements(16+i,3:4) = Elements(16+i,3:4) + [20 20];
end
% vertical columns
Elements(31,:) = [2   5       31 21  200];
Elements(32,:) = [2   5       30  1  200];
% duplicate the arch at y=h
for i=1:32,
    Elements(i+32,:) = Elements(i,:)+[0 0 100 100 0];
end
% add beams for crane rails
Elements(65,:) = [2   7       1   135 200];
Elements(66,:) = [2   7       135 101 200];
Elements(67,:) = [2   7       21  136 200];
Elements(68,:) = [2   7       136 121 200];
% add cross bar and beams
Elements(69,:) = [1   6       9  129  200];
Elements(70,:) = [1   6       29 109  200];
Elements(71,:) = [1   6       8  108  200];
Elements(72,:) = [1   6       28 128  200];
Elements(73,:) = [1   6       6  104  200];
Elements(74,:) = [1   6       26 124  200];
Elements(75,:) = [1   6       4  106  200];
Elements(76,:) = [1   6       24 126  200];

% plot the mesh

figure(1),plotmesh(Nodes,Elements)

%%%%%%%%% build elementary matrices and assemble  %%%%%%

% build index table for dof: locnod(i,:) = list of degrees of freedom number attached to node i
%
Nodes_active = unique(Elements(:,3:4));   % find which node really are used in elements. Finds the
unique elements.
                                          % the degrees of freedom fixed on the ground will
                                          % be removed later.
Ndof = size(Nodes_active,1)*6;            % 6 dof per node
locnod(Nodes_active,1:6) = reshape([1:Ndof],6,Ndof/6)';        % ':  transpose of matrix.
Nele   = size(Elements,1);
```

```
K = sparse(Ndof,Ndof);
M = sparse(Ndof,Ndof);

for el = 1:Nele,

   type   = Elements(el,1);
   matel  = Mat(Elements(el,2),:);      %materials for the specific element
   node1  = Nodes(Elements(el,3),:);  % coordinates of node 1
   node2  = Nodes(Elements(el,4),:);  % coordinates of node 2
   node3  = Nodes(Elements(el,5),:);  % coordinates of node 3 (for beams)

   % Build element matrices
   if type==1,      % bar
       % write the function yourself!!
      [Kel,Mel] = bar(node1,node2,matel);   %Kel = (4x4), Mel = (6x6)
         dof=[locnod(Elements(el,3),[1 2 3]) ...
              locnod(Elements(el,4),[1 2 3]) ] ; % only the translational
   elseif type==2,
      [Kel,Mel] = beam(node1,node2,node3,matel);
         dof=[locnod(Elements(el,3),:) ...              % ... means line break
              locnod(Elements(el,4),:) ] ;
   end

   %% Assemble
   K(dof,dof) = K(dof,dof)+Kel;
   M(dof,dof) = M(dof,dof)+Mel;



   clear matel type Kel Mel dof
end


%% PART 1 of assignment.
% Checking the model: Verify that, if no degrees of freedom are fixed,
% K * u_trans = 0
% where u_trans is a translation rigid body mode with unit amplitude.

% K: (264x264) matrix
% M: (264x264) matrix
% u_trans: (264x1) matrix

u_trans = zeros(size(K,1),1);
for i = 3:6:size(u_trans)
    u_trans(i) = 1;         % Setting u1 = u2 = 1 (one unit distance).
end

null_check = full(K) * u_trans;

disp('Max. value of null_check vector: ')
disp(max(null_check))       %Gives 7.4506e-09, which has to be within reasonable limits.



%% Checking the total mass of the structure by computing
% u_trans' * M * u_trans = m_total

m_total = u_trans' * M * u_trans;

%% Calculating the actual mass of the structure: m = rho * A * L
% We have 7 different bars and beams, all with different A and L.
% Following the naming convention given in drawing:

beam1 = 8 * Mat(1,3) * Mat(1,4) * l1;       %8 beam 1 elements with length l1. Mat(1,3): rho,
Mat(1,4): A
beam1 = beam1 + 4 * Mat(1,3) * Mat(1,4) * sqrt((2 * l1)^2);    % 4 beam1 elements at 45 degree angle.
```

```
beam2 = 8 * Mat(2,3) * Mat(2,4) * sqrt(l2^2 + l3^2);        % 8 lower beam2 elements
beam2 = beam2 + 8 * Mat(2,3) * Mat(2,4) * sqrt(l4^2 + l5^2);    % 8 middle beam2 elements
beam2 = beam2 + 4 * Mat(2,3) * Mat(2,4) * sqrt(l6^2 + l7^2);    % 4 top-top beam2 elements
beam2 = beam2 + 4 * Mat(2,3) * Mat(2,4) * sqrt((l7 - l1)^2 + l6^2);     % 4 top-lower beam2 elements

beam3 = 4 * Mat(3,3) * Mat(3,4) * l10;          %4 topmost beam3 elements
beam3 = beam3 + 4 * Mat(3,3) * Mat(3,4) * sqrt(l8^2 + l9^2);    % 4 beam3 elements with an angle


bar4 = 8 * Mat(4,3) * Mat(4,4) * l1;            % 8 horizontal bar4 elements with length l1
bar4 = bar4 + 4 * Mat(4,3) * Mat(4,4) * sqrt((l3 - l1)^2 + l2^2);       % 4 lowermost bar4 elements
bar4 = bar4 + 4 * Mat(4,3) * Mat(4,4) * sqrt((l5 - l1)^2 + l4^2);       % 4 uppermost bar4 elements

beam5 = 4 * Mat(5,3) * Mat(5,4) * H;        % 4 beam5 elements (foundation blocks)

bar6 = 2 * Mat(6,3) * Mat(6,4) * h;         % 2 vertical bar6 elements
bar6 = bar6 + 4 * Mat(6, 3) * Mat(6,4) * sqrt((l5^2 + h^2));            %4 outermost cross bar6
elements
bar6 = bar6 + 2 * Mat(6, 3) * Mat(6,4) * sqrt((2 * l10)^2 + h^2);

beam7 = 2 * Mat(7,3) * Mat(7,4) * h;         % 2 vertical beam7 elements

m_total_calc = beam1 + beam2 + beam3 + bar4 + beam5 + bar6 + beam7;

disp('Absolute value of difference between u_trans^T * M * u_trans and calculated total mass: ')
disp(abs(m_total_calc - m_total))

mass_error_percent = ((abs(m_total_calc - m_total)) / m_total) * 100;           %Currently at 0.65%.
Seems very reasonable.




%% Apply boundary conditions
% fix dofs of nodes 30 31 130 131
%a. find the degrees of freedom that are fixed
dof_fix = [];
for n=[30 31 130 131],
    dof_fix = [dof_fix  locnod(n,:) ];
end
Ndof_fix = size(dof_fix,2);
%b. remove these fixed dof from the list of dof and eliminate them in the matrices
dof_rem = setdiff([1:Ndof],dof_fix);% remaining degrees of freedom
Ndof_rem=Ndof-Ndof_fix;
K=K(dof_rem,dof_rem);   % from here on, K and M are related to the dof_remaining
M=M(dof_rem,dof_rem);

%% Compute eigensolutions
[X,omega2_eig] = eig(full(K),full(M));          % corresponds to K*X = M*X*Omega^2 (our original
problem Ax = 0).
                                        % [X, Omega2] = eig(K, M)
                                        % <==>   K*X = M*X*Omega2
                                        %


[omega2_eig,k] = sort(real(diag(omega2_eig)));      % extracting the real part of the diagonal
entries of omega2_eig
X = real(X(:,k));
clear k

omega_eig = sqrt(omega2_eig)./(2*pi);           % Gives all the eigenfrequencies. Converted to
Hertz.
```

```
%% PART 2: Dynamic analysis.
% First task: Free vibration
%Compute the first 10 eigenmodes and eigenfrequencies of the system (do not forget
%to apply the boundary conditions from now on). For that, program and apply the
%inverse iteration technique with de
%ation. Verify your algorithm by comparing the
%eigensolutions obtained with the eigensolutions computed from the eig or eigs func-
%tion of Matlab. In particular verify that the accuracy of higher modes is deteriorating
%due to the successive application of de
%ation.

disp('Rank of K: ')
rank(full(K))
disp('Rank of M: ')
rank(full(M))

% As rank(K) = rank(M) = size(M) = size(K) = 240, meaning that M and K consist of 240
% linear independent entries, making them both non-singular.

M_constrained = full(M);        % Mass matrix, with degrees of freedom set (240 x 240)
K_constrained = full(K);        % Stiffness matrix, with degrees of freedom set (240 x 240)



%% Applying the inverse iteration technique

no_eigenmodes = 10;
omega2_inv = zeros(no_eigenmodes, 1);
omega_inv = zeros(no_eigenmodes, 1);
z = ones(size(M,1), no_eigenmodes);         % z vector from inverse iteration, eq. (3.4.25)
x = ones(size(M,1), no_eigenmodes);         % eigenvectors. Size:(DOF x no_eigenmodes)
P = 1;              % Will be the projection operator. Must first be set to 1 for the first
eigenmode.
it_sum = 0;         %sum of all iterations. Used to find average number of iterations.
for k = 1:1:no_eigenmodes
    [z(:,k), P, iterations] = inv_iter(K_constrained, M_constrained, x, k, P);

    x(:,k) = z(:,k);
    omega2_inv(k,1) = ( x(:,k).' * K * x(:,k) ) / ( x(:,k).' * M * x(:,k) );       % Rayleigh
quotient (eq.3.4.15) [rad/s^2]
    omega_inv(k,1) = sqrt(omega2_inv(k,1)) ./ (2 * pi);                        % Converting from
[rad/s] to [Hz]

    it_sum = it_sum + iterations;
end
it_avg = it_sum / no_eigenmodes;
fprintf('----Average iterations of inv_iter-------\n')
fprintf('\nit_avg =  %d\n', it_avg);
fprintf('--------------------------------\n')

% Accuracy measurement by comparing eigenfrequencies found by inverse
% iteration to the ones found by eigs in MATLAB:
omega_accuracy = zeros(no_eigenmodes,1);
eigenmode_accuracy = zeros(no_eigenmodes,1);
for i = 1:1:no_eigenmodes
   omega_accuracy(i,1) = (omega_eig(i,1) - omega_inv(i,1)) / omega_eig(i,1)  * 100;
   eigenmode_accuracy(i,1) = max(K * x(:,i) - (omega2_inv(i,1) * M * x(:, i)));
end



%% Time integration with implicit Newmark scheme

%Setting initial values:
q0 = 0;         % initial displacement at t = 0
```

```
q0_dot = 0;         % initial velocity at t = 0

h_vec = logspace(0, -2, 3);
%h_vec = [8.957277139407033e-04, 5e-04, 1e-04];        %Used for computational analysis
cost_imp = zeros(size(h_vec,2), 3);         % Vector containing data for computational time. [h, t]

for i_h = 1:1:size(h_vec,2)
    tic
    h = h_vec(i_h);
    gamma_imp = 1/2;
    beta_imp = 1/4;
    C = zeros(size(M,1), size(M,2));

    [q_imp, q_dot_imp, q_dot_dot_imp, h, num_t_entries, t] = Newmark(gamma_imp, beta_imp, K, M, C, h,
q0, q0_dot, omega_inv(4,1));

    cost_imp(i_h, 1) = h;
    cost_imp(i_h, 2) = num_t_entries;
    cost_imp(i_h, 3) = toc;


    %% Plotting:
    % Element 10 consists of node 8-9, and dof 43-54. Dof 51 is therefore u3,
    % corresponding to displacement in the z direction in the node where the
    % force is applied.

    %% Setting variables used in every plot: %%%%%%%%%%%%%%%%%%%%%%%%%%%
    plotStyle = {'-.','--','-', ':'};
    output_Path = 'C:\Users\adria\Documents\Dokumenter\TUM\Structural Dynamics\Assignment\figures';
    width = 175;                % width of exported plot figure
    height = 100;                   % height of exported plot figure
    lineSize = 2;                 %linesize of plotted lines (linewidth)
    titleSize = 30;
    labelSize = 22;
    valueSize = 100;         %Must be this high to get pdf export right. Originally at 20.
    legendInfo{i_h} = strcat('$h = $', num2str(h), 's');            %
https://se.mathworks.com/matlabcentral/answers/29799-adding-a-legend-and-different-coloured-lines-in-
a-for-loop

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    figure(2)
    plot(t, q_imp(51,:), 'k', 'Linestyle', plotStyle{i_h})
    title({'Displacement in Z direction of node subjected to', 'triangular force for varying step
size'}, 'Fontsize', titleSize)
    xlabel('Time, $t$  $[s]$', 'FontSize', labelSize, 'Interpreter', 'Latex')
    ylabel('${q}$        $[m]$', 'FontSize', labelSize, 'Interpreter', 'Latex')
    set(gca,'FontSize', valueSize);
    set(gcf, 'PaperPosition', [0 0 width height])
    set(gcf, 'PaperSize', [width height])
    grid on
    grid minor
    hold on

    figure(3)
    plot(t, q_dot_imp(51,:), 'k', 'Linestyle', plotStyle{i_h})
    title({'Velocity in Z direction of node subjected to', 'triangular force for varying step size'},
'Fontsize', titleSize)
    xlabel('Time, $t$  $[s]$', 'FontSize', labelSize, 'Interpreter', 'Latex')
    ylabel('$\dot{q}$       $[\frac{m}{s}]$', 'FontSize', labelSize, 'Interpreter', 'Latex')
    set(gca,'FontSize', valueSize);
    set(gcf, 'PaperPosition', [0 0 width height])
    set(gcf, 'PaperSize', [width height])
    grid on
    grid minor
    hold on
```

```
    figure(4)
    plot(t, q_dot_dot_imp(51,:), 'k', 'Linestyle', plotStyle{i_h}, 'Linewidth', 1.3)
    title({'Acceleration in Z direction of node subjected to', 'triangular force for varying step
size'}, 'Fontsize', titleSize)
    xlabel('Time, $t$  $[s]$', 'FontSize', labelSize, 'Interpreter', 'Latex')
    ylabel('$\ddot{q}$     $[\frac{m}{s^2}]$', 'FontSize', labelSize, 'Interpreter', 'Latex')
    set(gca,'FontSize', valueSize);
    set(gcf, 'PaperPosition', [0 0 width height])
    set(gcf, 'PaperSize', [width height])
    grid on
    grid minor
    hold on

end
%Legends and exporting must be applied afterwards due to the for loop
figure(2)
legend(legendInfo, 'Location', 'southeast', 'Interpreter', 'latex')
saveas(figure(2), fullfile(output_Path, 'var_h_disp'), 'pdf')

figure(3)
legend(legendInfo, 'Location', 'southeast', 'Interpreter', 'latex')
saveas(figure(3), fullfile(output_Path, 'var_h_vel'), 'pdf')

figure(4)
legend(legendInfo, 'Location', 'southeast', 'Interpreter', 'latex')
saveas(figure(4), fullfile(output_Path, 'var_h_acc'), 'pdf')



%% Explicit time integration - central difference
% Defining all needed matrices
gamma_exp = 1/2;
beta_exp = 0;

h_stable = 2 / sqrt(max(omega2_eig));

% Part used for calculation costs:
%{
h_vec = [8.957277139407033e-04, 5e-04, 1e-04];        %REMOVE after computational analysis
cost_exp = zeros(size(h_vec,2), 3);         % Vector containing data for computational time. [h, t]

for i_h = 1:1:size(h_vec,2)
    tic
    h = h_vec(1,i_h)
    [q_exp_stable, q_dot_exp_stable, q_dot_dot_exp_stable, h_exp_stable, num_t_entries, t_exp_stable]
= Newmark(gamma_exp, beta_exp, K, M, C, h, q0, q0_dot, omega_inv(4,1));
    cost_exp(i_h, 1) = h_exp_stable;
    cost_exp(i_h, 2) = num_t_entries;
    cost_exp(i_h, 3) = toc;
end
%}

[q_exp_stable, q_dot_exp_stable, q_dot_dot_exp_stable, h_exp_stable, num_t_entries, t_exp_stable] =
Newmark(gamma_exp, beta_exp, K, M, C, h_stable, q0, q0_dot, omega_inv(4,1));
[q_exp_unstable, q_dot_exp_unstable, q_dot_dot_exp_unstable, h_exp_unstable, num_t_entries,
t_exp_unstable] = Newmark(gamma_exp, beta_exp, K, M, C, 1.01*h_stable, q0, q0_dot, omega_inv(4,1));


figure(5)
subplot(2,1,1)
plot(t_exp_stable, q_exp_stable(51, :), 'k', 'Linewidth', lineSize)
title({'Stable explicit Newmark integration', 'with the central difference algorithm'}, 'Fontsize',
titleSize)
xlabel('Time, $t$  $[s]$', 'FontSize', labelSize, 'Interpreter', 'Latex')
ylabel('${q}$       $[m]$', 'FontSize', labelSize, 'Interpreter', 'Latex')
h_legend = legend('$h = h_{stable}$' , 'Location', 'south');     % must be done like this...
https://se.mathworks.com/matlabcentral/newsreader/view_thread/254118
```

```
set(h_legend, 'Interpreter', 'Latex')
set(gca,'FontSize', valueSize);
set(gcf, 'PaperPosition', [0 0 width height])
set(gcf, 'PaperSize', [width height])
grid on
grid minor


subplot(2,1,2)
plot(t_exp_unstable, q_exp_unstable(51, :), 'k', 'Linewidth', lineSize)
title({'Unstable explicit Newmark integration', 'with the central difference algorithm'}, 'Fontsize',
titleSize)
xlabel('Time, $t$  $[s]$', 'FontSize', labelSize, 'Interpreter', 'Latex')
ylabel('${q}$       $[m]$', 'FontSize', labelSize, 'Interpreter', 'Latex')
h_legend = legend('$h = 1.01 \cdot h_{stable}$', 'Location', 'south');
set(h_legend, 'Interpreter', 'Latex')
set(gca,'FontSize', valueSize);
set(gcf, 'PaperPosition', [0 0 width height])
set(gcf, 'PaperSize', [width height])
grid on
grid minor

saveas(figure(5), fullfile(output_Path, 'exp_Newmark'), 'pdf')



%% Plotting displacements of the hangar

figure(6)           % Plotting the computed displacement from the applied load
q_plot = zeros(Ndof,Ndof_rem);
q_plot(dof_rem, 1) = q_exp_stable(:,1000);
plotmesh_no_nodes(Nodes,Elements);
plotdis(Nodes,Elements,locnod,q_plot(:,1), 500);
set(gcf, 'PaperPosition', [0 0 width height])
set(gcf, 'PaperSize', [width height])
saveas(figure(6), fullfile(output_Path, 'force_displacement'), 'pdf')



%% Setting variables used in 10 eigenmodes plot: %%%%%%%%%%%%%%%%%%%%%%%%%%%%
plotStyle = {'-.','--','-', ':'};
output_Path = 'C:\Users\adria\Documents\Dokumenter\TUM\Structural Dynamics\Assignment\figures';
width_eig = 100;                % width of exported plot figure
height_eig = 175;                   % height of exported plot figure
lineSize = 2;                 %linesize of plotted lines (linewidth)
titleSize = 30;
labelSize = 22;
valueSize = 100;         %Must be this high to get pdf export right. Originally at 20.
legendInfo{i_h} = strcat('$h = $', num2str(h), 's');            %
https://se.mathworks.com/matlabcentral/answers/29799-adding-a-legend-and-different-coloured-lines-in-
a-for-loop

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


Xplot=zeros(Ndof,Ndof_rem); % express the modes in the non-fixed numbering used in locnod
Xplot(dof_rem,:)=X;

figure(7) % all 10 firstcomputed eigenmodes X
for i = 1:1:10
    subplot(5,2,i)
    plotdis(Nodes,Elements,locnod,Xplot(:,i),5);
end

set(gcf, 'PaperPosition', [0 0 width_eig height_eig])
set(gcf, 'PaperSize', [width_eig height_eig])
```

```
saveas(figure(7), fullfile(output_Path, 'eig_plot'), 'pdf')
% If this section only is run (mark and F9), the plot figures all get the
% same size, which it doesn't if you run the whole script.
```

F.1 The inv_iter function

```
function [ z, P, iterations_actual ] = inv_iter( K, M, x, k, P)
    z0 = ones(size(M,1),1);          %Starting vector.
    z0 = z0 / norm(z0);     %normalizing starting vector.
    tol = 0.0001;             % accepted convergence accuracy tolerance.
    epsilon = 1;            % error measurement


    if k == 1       % if first eigenmode.
        P = eye(size(M,1));         % Projection operator set to identity matrix

    else
        P = P * (eye(size(M,1)) - (x(:, k - 1) * x(:, k - 1).' * M ) / (x(:, k - 1).' * M * x(:, k -
1)));
    end

    [K_L, K_U] = lu(K);         % Factorizing K to a lower and upper matrix.
    z = z0;     %z is initialized with the normalized, random vector.
    iterations_actual = 0;
    %for p = 1:1:iterations
        while epsilon >= tol
            omega2_est_0 = (z.' * K * z) / (z.' * M * z);       % estimating Rayleigh quotient
(3.4.15)
            y = M * z;              % eq. (3.4.23)
            b = P.' * y;                % want to solve Kz = P^T*y (3.4.24) with forward/backward
substitution.
            b_hat = K_L \ b;            % forward substitution (2.1.9)
            z_hat = K_U \ b_hat;        % backward substitution (2.1.10)
            z = P * z_hat;          % Projecting z on to the correct dimension.
            %z_hat = K \ P.' * y;
            %z = P * z_hat;

            z = z / norm(z);     % Normalizing z
            omega2_est_1 = (z.' * K * z) / (z.' * M * z);       % estimating Rayleigh quotient at p +
1 (3.4.15)
            epsilon = norm(omega2_est_1 - omega2_est_0);        % updating errror measurement.
            iterations_actual = iterations_actual + 1;
        end
    %end

    fprintf('----Results of inv_iter-------\n')
    fprintf('\nEigenmode no. %d\nEpsilon: %d\nIterations: %d\nz(1,1): %d\n', k, epsilon,
iterations_actual, z(1,1));
    fprintf('--------------------------------\n')

end
```

F.2 The Newmark function

```
function [q, q_dot, q_dot_dot, h, num_t_entries, t] = Newmark(gamma, beta, K, M, C, h, q0, q0_dot,
omega)
% Creating vector p, containing the triangular force subjected to the
% structure and time vector. Collected in function for easy reuse.

deltaT = 10.5;            %Time after the force is applied [s]


[p, num_t_entries, t] = createForceAndTime(h, omega, 100, deltaT );
p_new = zeros(size(M,1), num_t_entries);            % The force must be a coloumn vector, with force
only in the node subjected to force (node 51)
p_new(51,:) = p';
p = -p_new;


% Defining all needed matrices
q_star = zeros(size(K,1), num_t_entries);        % q of size (1 x n)
q = zeros(size(K,1), num_t_entries);        % q of size (1 x n)
q(1,1) = q0;

q_dot_star = zeros(size(K,1), num_t_entries);        % q_dot of size (1 x n)
q_dot = zeros(size(K,1), num_t_entries);        % q_dot of size (1 x n)
q_dot(1,1) = q0_dot;

q_dot_dot = zeros(size(K,1), num_t_entries);        % q_dot_dot of size (1 x n)
%q_dot_dot_imp(1,1) = q0_dot_dot_imp(1,1);

S = M + h * gamma * C + h^2 * beta * K;
[S_L, S_U] = lu(S);


for i = 2 : 1 : num_t_entries       % TODO: change to half the period of the fourth eigenfrequency.
    % Time incrementation (done at the start of every for loop)

    % Prediction
    q_dot_star(:, i) = q_dot(:, i - 1) + ((1 - gamma) * h * q_dot_dot(:, i - 1));
    q_star(:,i) = q(:, i - 1) + (h * q_dot(:, i - 1)) + (0.5 - beta) * h^2 * q_dot_dot(:, i - 1);

    % Acceleration computing
    % S, S_L and S_K is computed before the for loop.

                                    % Solving q_dot_dot(:,i) = S \ ( p(i,1)
                                    % - K * q_star(:,i)) with LU
                                    % factorization
    b = p(:,i) - (C * q_dot_star(:,i)) - K * q_star(:,i);
    b_hat = S_L \ b;        %Forward substitution
    q_dot_dot(:,i) = S_U \ b_hat;           %Backwards substitution.

    % Correction
    q_dot(:, i) = q_dot_star(:,i) + h * gamma * q_dot_dot(:,i);
    q(:,i) = q_star(:,i) + h^2 * beta * q_dot_dot(:, i);
end

fprintf('----Results of Newmark-------\n')
fprintf('\nno_t_entries: %d\nh: %d\n', num_t_entries, h);
fprintf('--------------------------------\n')


end
```

F.3 The createForceAndTime function

```
function [p, num_t_entries, t] = createForceAndTime(h, omega, F_max, deltaT)

T = 1 / 2 * (1 / omega);       % The period of the force
num_t_entries = int64((T + deltaT) / h);

t = zeros(num_t_entries, 1);
p = zeros(num_t_entries, 1);

for i = 1 : 1 : size(t,1)
    if i ~= num_t_entries
        t(i + 1,1) = t(i,1) + h;
    end

    if (t(i,1) <= (T / 2))
        p(i,1) = F_max / (T/2) * t(i, 1);
    elseif ((t(i,1) > T/2) && (t(i,1) <= T ))
        p(i,1) = - F_max / (T/2) * t(i, 1) + 2 * F_max;
    else
        p(i,1) = 0;
    end
end
```

F.4 The bar function

```
%%%%%%%%%% function for building bar elements  %%%%%%%%%%
function [Kel,Mel] = bar(node1,node2,mat),

%  node1=[x1 y1 z1];
%  node2=[x2 y2 z2];
%  mat  =[E G m A Ix  Iy  Iz]

l = norm(node1-node2);
E = mat(1);
A = mat(4);
rho = mat(3);

Kel_local  = E*A/l*[ 1 -1;
                    -1  1];

Mel_local  = rho*A*l/6*[2 0 0 1 0 0;
                        0 2 0 0 1 0;
                        0 0 2 0 0 1;
                        1 0 0 2 0 0;
                        0 1 0 0 2 0;
                        0 0 1 0 0 2];

e_x = (node2-node1)/l;

T   = [e_x 0 0 0; 0 0 0 e_x];
Kel = T' * Kel_local *T;    % (4x4) matrix
Mel = Mel_local ;       % (6x6) matrix
```

F.5 The beam function

```
function [Kel,Mel] = beam(node1,node2,node3,mat)

%  node1=[x1 y1 z1];
%  node2=[x2 y2 z2];
%  node3=[x3 y3 z3]; for Ix direction
%  mat  =[E G m A Ix  Iy  Iz]

l = norm(node1-node2);          %Absoluttverdi
if l==0,
    disp('zero length')
    node1
    node2
    return
end
E = mat(1);
G = mat(2);
rho = mat(3);
A = mat(4);
J_x = mat(5);
I_y = mat(6);
I_z = mat(7);
r2  = (I_y+I_z)/A;

Kel_local  =  zeros(12,12);             %six degrees of freedom * 2. Matrix is in page 35 in lecture notes

Kel_local([1 7],[1 7]) = E*A/l*[ 1 -1;
                                -1  1];
Kel_local([2 6 8 12],[2 6 8 12]) = E*I_z/l^3* ...
  [12   6*l   -12   6*l;
   6*l  4*l^2 -6*l  2*l^2;
   -12  -6*l  12    -6*l;
   6*l  2*l^2 -6*l  4*l^2];

Kel_local([3 5 9 11],[3 5 9 11]) = E*I_y/l^3* ...
  [12   -6*l   -12   -6*l;
   -6*l  4*l^2 6*l  2*l^2;
   -12   6*l   12    6*l;
   -6*l  2*l^2 6*l  4*l^2];

Kel_local([4 10],[4 10]) = G*J_x/l*[ 1 -1;
                                    -1  1];

Mel_local  =  zeros(12,12);

Mel_local([1 7],[1 7]) = rho*A*l/6*[ 2 1;
                                     1 2];
Mel_local([2 6 8 12],[2 6 8 12]) = rho*A*l/420* ...
   [156   22*l   54   -13*l;
    22*l  4*l^2  13*l -3*l^2;
    54    13*l   156  -22*l;
   -13*l  -3*l^2 -22*l 4*l^2];

Mel_local([3 5 9 11],[3 5 9 11]) = rho*A*l/420* ...
   [156   -22*l   54   13*l;
    -22*l  4*l^2  -13*l -3*l^2;
    54    -13*l   156  22*l;
    13*l  -3*l^2  22*l 4*l^2];
Mel_local([4 10],[4 10]) = rho*A*l*r2/6*[ 2 1;
                                         1 2];

e_x = (node2-node1)/l;
d2  = node2-node1;
d3  = node3-node1;
e_y = cross(d3,d2)/norm(cross(d3,d2));
e_z = cross(e_x,e_y);           %cross product

R = [ e_x(1) e_x(2) e_x(3);
      e_y(1) e_y(2) e_y(3);
      e_z(1) e_z(2) e_z(3)];
T = zeros(12,12);
T([1:3],[1:3]) = R;
T([4:6],[4:6]) = R;
T([7:9],[7:9]) = R;
T([10:12],[10:12]) = R;

Kel = T'*Kel_local*T;
Mel = T'*Mel_local*T;
```

F.6 The plotdis function

```
function  outputstring=plotdis(Nodes,Elements,locnod,disp,ampl),

% plots the mesh with displacement multiply by ampl

Nel = size(Elements,1);

%plot3(Nodes(:,1),Nodes(:,2),Nodes(:,3),'g*')
for el=1:Nel,
    type   = Elements(el,1);
    hold on
    dof1 = locnod(Elements(el,3),[1 2 3]);
    dof2 = locnod(Elements(el,4),[1 2 3]);
    %add the displacements to the coordinates of the nodes of the element
    node1  = Nodes(Elements(el,3),:) + disp(dof1)'*ampl;
    node2  = Nodes(Elements(el,4),:) + disp(dof2)'*ampl;
    if type==1,
        plot3([node1(1);node2(1)],[node1(2);node2(2)],[node1(3);node2(3)],'k', 'Linewidth', 2);
    elseif type==2,
        plot3([node1(1);node2(1)],[node1(2);node2(2)],[node1(3);node2(3)],'k', 'Linewidth', 2);
    end
    clear dof1 dof2
end
axis equal
axis off
view(-40,15)
```

F.7 The plotmesh_no_nodes function

```
function  outputstring=plotmesh_no_nodes(Nodes,Elements),

Nel = size(Elements,1);

clf;        %Clears the current figure
%plot3(Nodes(:,1),Nodes(:,2),Nodes(:,3),'k*')
for el=1:Nel,
   type   = Elements(el,1);
   node1  = Nodes(Elements(el,3),:);
   node2  = Nodes(Elements(el,4),:);
   hold on
   if type==1,              % if bar
      plot3([node1(1);node2(1)],[node1(2);node2(2)],[node1(3);node2(3)],':k', 'Linewidth', 2);
   elseif type==2,          %if beam
      plot3([node1(1);node2(1)],[node1(2);node2(2)],[node1(3);node2(3)],':k', 'Linewidth', 2);
   end
end
axis equal
%axis off
view(-65,35)
```