Mobility & Built Environment
Molengraaffsingel 8
2629 JD Delft
P.O. Box 49
2600 AA Delft
The Netherlands

www.tno.nl

T +31 88 866 22 00

TNO report

TNO 2024 R11057A | Final

V2004-WP3.1: OpenRadioss as a reliable replacement for LS-DYNA

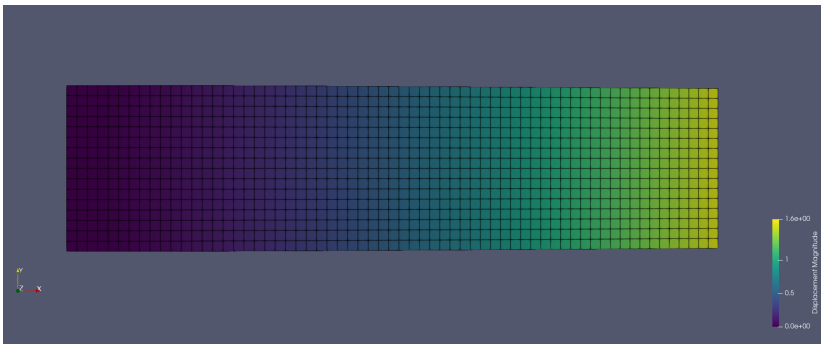
Date	April 23, 2025
Author(s)	F.S.J. Nobels M.Sc.
Classified by	ir J.A.A. Vaders
Classification date	25-10-2024 This classification will not change
Title	Ongerubriceerd, releasable to the public
Management summary	Ongerubriceerd, releasable to the public
Report text	Ongerubriceerd, releasable to the public
Appendices	Ongerubriceerd, releasable to the public
Copy no	
No. of copies	2
Number of Pages	120 (incl. appendices excl. distribution list)
Number of appendices	3
Sponsor	COMMIT
Project name	V2004 WP3.1 Overall UNDEX simulatietool
Project number	060.43015/01.01

The classification designation Ongerubriceerd is equivalent to Unclassified, Departementaal Vertrouwelijk is equivalent to Restricted, Stg. Confidentieel is equivalent to Confidential and Stg. Geheim is equivalent to Secret.

MANAGEMENTUITTREKSEL TNO-RAPPORT: TNO 2024 R11057A

V2004-WP3.1: OpenRadioss as a reliable replacement for LS-DYNA

In V2004 heeft TNO het gebruik onderzocht van de open source eindige-elementenmethode (e.e.m.) pakket OpenRadioss als een (gedeeltelijke) vervanging voor LS-DYNA. OpenRadioss kan worden gebruikt om de (zeer) dynamische en niet-lineaire structurele responsie te bepalen van constructies onder belasting.



Probleemstelling

Wij ontwikkelden de onderwater schok code 3DCAV, 3DCAV gebruikt LS-DYNA als een e.e.m. pakket om de structurele responsie te bepalen van schepen. In 2019 is LS-DYNA overgenomen door Ansys. De hiermee gepaard gaande hogere kosten in combinatie met enkele andere factoren hebben ertoe geleid dat besloten is om naar alternatieven te kijken. In 2022 is Radioss open source geworden onder de naam OpenRadioss. OpenRadioss heeft geen licensiekosten. Aanvullend, OpenRadioss is open source, dus is er een gemeenschap van ontwikkelaars waarmee we rechtstreeks contact kunnen hebben. Het doel van dit rapport is onderzoeken of LS-DYNA (gedeeltelijk) vervangen kan worden door OpenRadioss.

Beschrijving van de werkzaamheden

We hebben onderzocht hoe je OpenRadioss op servers kunt compileren en gebruiken. We hebben standaard benchmarktests ontwikkeld om de nauwkeurigheid van OpenRadioss te verifiëren. Specifiek, ontwikkelen we twee sets benchmarkstesten voor beam-/shellelementen en veerelementen. Ten eerste, ontwikkelen en gebruiken we een cantilever-test om shell- en beamelementen te verifiëren. Specifiek hebben we een resolutietest met verschillende numerieke resoluties uitgevoerd en vergeleken de numerieke oplossing met de analytische oplossing. Ten tweede ontwikkelen we voor de veerelementen één veerelementbenchmark met een opgelegde kracht of opgelegde snelheid om te verifiëren of veerelement zich correct gedragen. Verder, vergelijken we de resultaten van deze benchmarks met de analytische oplossing. Bovendien gebruiken we deze benchmarks ook om de prestaties van high-performance computing (HPC) te onderzoeken voor OpenRadioss. Specifiek vergelijken we verschillende parallelisatiestrategieën en bepalen de ideale strategie op onze servers.

Resultaten en conclusies

We hebben verschillende structurele dynamische benchmarkproblemen gemaakt die we hebben gebruikt om shell, beam en eenvoudige veerelementen te verifiëren. Tijdens het werken aan OpenRadioss werden een paar problemen gevonden en de ondersteuning van de ontwikkelaars van OpenRadioss was uitstekend. We hebben goede prestaties gevonden voor beam, shell, en veerelementen. Wij raden aan om verder te onderzoeken of OpenRadioss een goede (gedeeltelijke) vervanging voor LS-DYNA is.

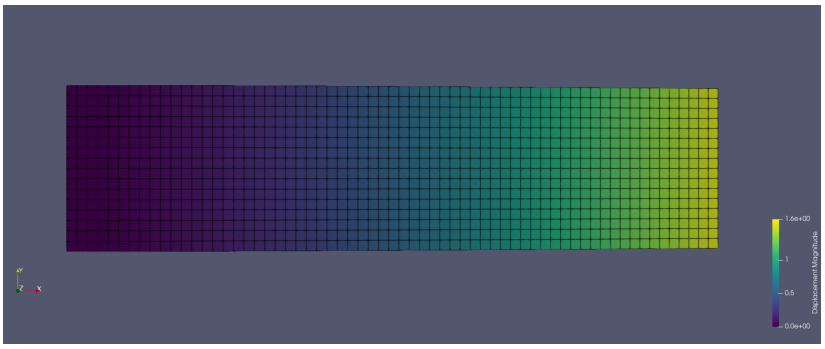
Toepasbaarheid

Dit werk is een eerste stap om in de toekomst meer en nauwkeuriger simulaties te doen met OpenRadioss van de responsie van onderwaterschokken en bovenwater dreigingen voor schepen en onderzeeboten.

MANAGEMENT SUMMARY TNO-REPORT: TNO 2024 R11057A

V2004-WP3.1: OpenRadioss as a reliable replacement for LS-DYNA

Within V2004, TNO investigated the use of the open source finite element method (FEM) package OpenRadioss as a (partial) replacement for LS-DYNA. OpenRadioss can be used to determine the (highly) dynamic and non-linear structural response of structures under loads.



Problem description

We developed the UNDEX (UNDerwater EXplosion) code 3DCAV, 3DCAV uses LS-DYNA as a FEM package to determine the structural response of ships. In 2019, LS-DYNA was acquired by Ansys. This has resulted in increased costs of licenses. These increased costs together with some other issues have led to the decision to investigate other options. In 2022, Radioss has become open source as OpenRadioss, and therefore OpenRadioss does not have license costs. Additionally, OpenRadioss is open source thus there is a community of developers that we can be in direct contact with. The goal of this report is to investigate if LS-DYNA can be replaced with OpenRadioss.

Work performed

We investigated how to compile and use OpenRadioss on servers. Furthermore, we developed standard benchmark tests to verify the accuracy of OpenRadioss. Specifically, we developed two sets of benchmark tests for beam/shell elements and spring elements. Firstly, we developed and used a cantilever beam test to verify shell and beam elements. Specifically, we performed a resolution test at different numerical resolutions and compared the numerical solution with the analytical solution. Secondly, for spring elements we developed and used a single spring element benchmark with an imposed force or imposed velocity to verify the accuracy of spring elements. We compared the results of these benchmarks with the analytical solution. Moreover, we also used these benchmarks to investigate the high-performance computing (HPC) performance of OpenRadioss. Specifically, we compared different parallelisation strategies and determined the ideal strategy on our servers.

Results and conclusions

We created several structural dynamics benchmark problems which we used to verify shell, beam and simple spring elements. While working on OpenRadioss a few issues were found and the support from the developers of OpenRadioss was excellent. We found good performance for beam, shell and spring elements. We recommend continuing investigating OpenRadioss as a replacement for LS-DYNA.

Applicability

This work is a first step towards doing larger number of and more accurate simulations with OpenRadioss in the future of the response to underwater shocks and above-water threats for ships and submarines.

Contents

1 Introduction

The aim of this report is to investigate OpenRadioss as a (partial) alternative for LS-DYNA. LS-DYNA is intensively used in the Naval & Offshore Structures department of TNO (TNO-NOS) to do structural analysis. LS-DYNA is used to calculate mainly the structural response of highly non-linear and dynamic phenomena. The two main motivations for changing from LS-DYNA to OpenRadioss are firstly the fact that OpenRadioss is open source which means no licenses are required and the code is accessible by TNO (without restrictions). Secondly, OpenRadioss has developers supporting users and an open-source community where questions and problems can be asked and discussed. The investigation in this report is mainly focused on the different element formulations and applying these to simulations for which an analytical solution is known. This is done to verify OpenRadioss and find any potential problems and bugs such that they can be fixed and made aware of. While investigating OpenRadioss the support of the OpenRadioss community is also considered.

1.1 Historical perspective of LS-DYNA and OpenRadioss

To get an impression of the difference and relation between LS-DYNA and OpenRadioss a short overview of the history of both codes is given. Fig. 1.1 shows the DYNA family tree. Originally, Dyna3d was developed to simulate the impact of the Full-Fuzzing-Option (FUFO) or 'Dial-a-yield' nuclear bombs that would be released at low altitudes. Because of the complicated physics of explosions, a full 3D simulation code would be required to accurately simulate the non-linear dynamics using explicit time integration. The original FUFO bomb (B77) was cancelled because its cost overrun, however, some features developed using Dyna3d were later implemented in the B83 bomb. Despite the cancellation of the FUFO bomb the development of Dyna3d continued. Dyna3d quickly became a dominant code, and its source code was released in the public domain without restrictions upon request from France (Benson, 2007). After this the development of Dyna3d continued and its main developer John Hallquist consulted over 60 different companies on how to use Dyna3d. After a change in policy of the Lawrence Livermore National Laboratory (LLNL) John Hallquist left and founded Livermore Software Technology Corporation (LSTC). LSTC started releasing LS-DYNA and the source code of LS-DYNA was no longer released as compared to Dyna3d (Benson, 2007).

After the release of the source code of Dyna3d, the ESI group was founded by Alain de Rouvray, Jacques Dubois, Iraj Farhoomand and Eberhard Haug. The ESI group developed Pam-Crash and used it to simulate the crash of a military fighter plane into a nuclear power plant (Haug, 1981) and for the first successful crash simulation of a frontal impact of a passenger car within a single night on a computer cluster (Haug et al., 1986). After demonstrating the capabilities of Pam-Crash it has been used widely in the automotive industry. Solanki et al. (2004) compared Pam-Crash and LS-DYNA and found that the differences between both codes are minimal for car crash analysis. Based on Pam-Crash, Radioss was developed and again widely used for crash simulation analysis. Altair (2022c) announced that Radioss will be made open source under the name of OpenRadioss. Altair remains to release the commercial Radioss parallel with OpenRadioss because three keywords of Radioss are not supported in OpenRadioss. These keywords are related to encryption and finite volume method airbags (Sharp, 2023c). These features are mainly

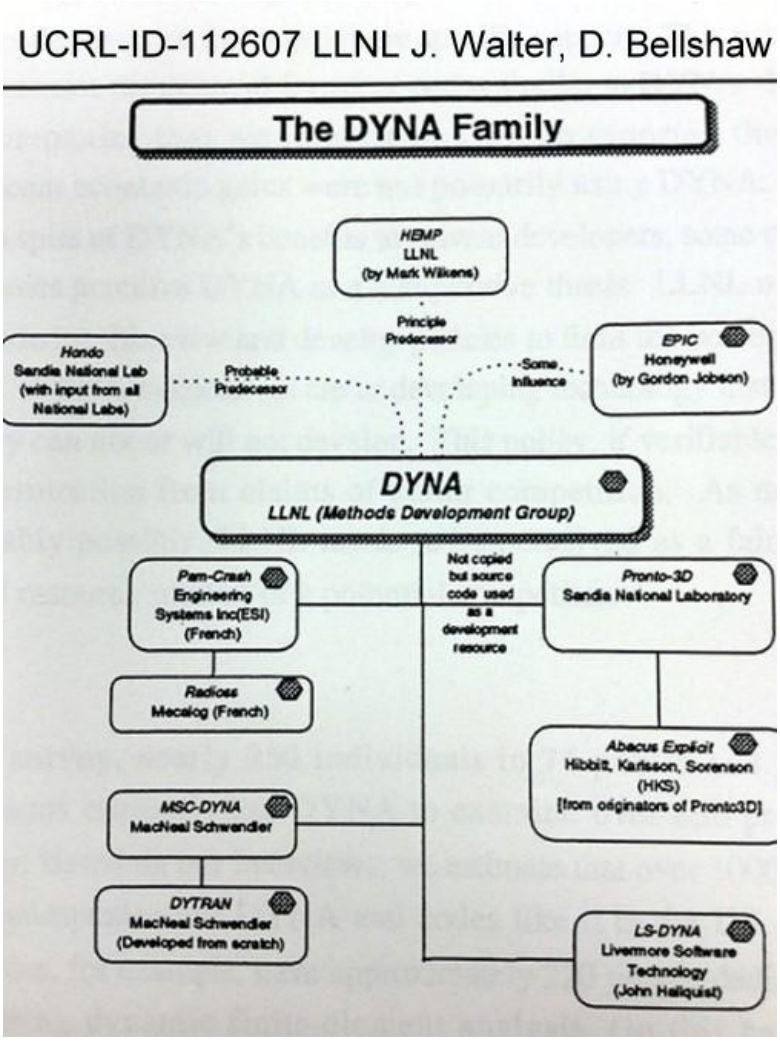


Figure 1.1 The DYNA family tree with the original DYNA3D code and its derived codes PAM-CRASH, later RADIOSS and OPENRADIOSS, PRONTO-3D later ABACUS explicit, MSC-DYNA later DYTRAN, and LS-DYNA. Many common explicit FEM codes are based on DYNA3D. Both LS-DYNA and OPENRADIOSS are based on the same original code of DYNA3D (Walter and Bellshaw, 1993).

used in the automotive industry for airbags and dummy models, so these will not be used much or at all in the TNO-NOS department.

This historical perspective of both LS-DYNA and OPENRADIOSS demonstrates that both codes are based on an identical base code (DYNA3D). This means that the basic structure of LS-DYNA and OPENRADIOSS is similar. Despite this, years of development has resulted in codes with different features and functionalities. The main aim of this report is to investigate how well OPENRADIOSS as a code performs, understand how to run it most efficient, and to understand the current limitations and advantages of using OPENRADIOSS as a possible partial replacement for LS-DYNA. For the remainder of this report the same name will be used for both RADIOSS and OPENRADIOSS because they are practically the same code for TNO-NOS.

1.2 Comparison of OpenRadioss and LS-DYNA

The literature lacks many direct comparisons between OpenRadioss and LS-DYNA, this means that the agreement between many features of OpenRadioss and LS-DYNA is not directly known. In the Pam-Crash time some comparisons have been done between LS-DYNA and Pam-Crash. Solanki et al. (2004) compared both for a frontal car crash and found that the damage between both codes is not the same and small differences exist in the deformation at the front of the car. Similarly, the internal energy has a slight different evolution, the differences between both FEM solvers are on the 5 to 10 per cent limit.

More recently, direct comparisons have been made between OpenRadioss and LS-DYNA. Di Pasquale (2015) found that the simulation of a crushed beam gives almost identical results in OpenRadioss and LS-DYNA. Cina (2019) looked at the difference for composite seats using crash analysis, but unfortunate his results are not publicly available, the library says that this thesis is classified as confidential, and they do not know the limitation period. Furthermore, Tofrowaih et al. (2021) used roof crush resistance tests and found that the difference between both solvers is small. Recently, Jezdik et al. (2023) simulated a tram collision with a crash dummy. They found that the simulations with OpenRadioss and LS-DYNA roughly agree with each other, however, the exact acceleration differs from experiments for both. Neither of the solvers performs significantly better than the other. Furthermore, Bini Leite et al. (2021) looked at a plane crash in the Hudson river and found that OpenRadioss and LS-DYNA give similar results. Interestingly, OpenRadioss conserved energy almost exactly, while LS-DYNA loses some four per cent of the total energy.

Overall, the impression from the literature is that both solvers are not too far off from each other and have a similar (dis)agreement with experiments.

1.3 Recent developments and uses of OpenRadioss

The use cases and recent developments of OpenRadioss are extremely diverse. Here a short overview of the most recent use cases and developments of OpenRadioss will be summarised. These developments are inside OpenRadioss and publicly available. Similarly, the FEM models of these recent use cases are freely available online.

OpenRadioss is used for simple models, Mestres (2023b) used OpenRadioss to model the bolts between two plates under rotation and displacement of one of the plates and see when the bolt fails. On a similar scale, Mestres (2023a) modelled the welding of two steel plates including the melting of the metal. He measured the resulted stress and temperature in the plate due to the welding. Nakano (2023) used OpenRadioss to model foam materials and tested foam blocks to see if they produce the behaviour of foam as expected, these foam models can directly be used in car crashes. On the other hand, Pasligh et al. (2017) modelled and tested a simplified model for rivets in OpenRadioss that use a calibrated cohesive element characteristics method which does not require to model the rivets and bolts in detail. This method can be used in large models for example car crash simulations.

OpenRadioss contains several examples from the car crash industry. The most known example is the Toyota Yaris crash test on a pole with 40 km/h (Sharp, 2023d). But also a simple bumper beam impacting on a single pole (Sharp, 2023b).

With the recent development of electric vehicles (EVs), Bulla et al. (2021) developed a material model for the separators of Lithium-Ion batteries and tested their model

under high mechanical loads and found good agreement in three different directions for their material model for separators. Bulla et al. (2023) expanded this investigation and simulated complete Lithium-Ion batteries under high mechanical loads and found good agreement with experiments. Shamchi et al. (2024) showed that the agreement between the simulations and experiments is good but at high displacements the pressure of the separators is slightly overestimated.

Furthermore, simulations to design protective structures (PS) have been performed with OPENRADIOSS. Prashanth (2022) tested a roll over protective structures (ROPS), similar Mittal et al. (2023) developed a new roll cage design for an all-terrain vehicle. They iteratively performed simulations using OPENRADIOSS and improved the design based on the maximum stresses found in the design. Brandão (2023) performed simulations on a falling object protective structure (FOPS), he specifically reproduced the standard ISO 3449 test to test the FOPS. This shows that performing standard ISO tests in OPENRADIOSS is possible.

Tests in the air have also been performed with OPENRADIOSS. Sharp (2023a) used OPENRADIOSS to simulate a bird strike on the windshield of an aeroplane. In this case the bird was modelled as a cylinder of smoothed-particle hydrodynamics (SPH) particles. Similarly, Franke et al. (2022) modelled the impact of a drone strike on aeroplane wings and windows. They used this to recommend improvements on the aeroplane design, especially, they recommended thicker windows to prevent too much damage from a drone strike.

Other threats from the air (AIREX) can also be simulated in OPENRADIOSS. Loverini and Robert (2022) used OPENRADIOSS to simulate air burst threats and use this to assess the survability of the target. They simulate this in two stages, the first stage was the use of a mirrored 2D simulation to simulate the detonation of 250 kg TNT including the reflection from the ground. The air was simulated with an ideal gas equation of state (EoS) while the explosive was simulated using the Jones-Wilkins-Lee (JWL) EoS (Jones and Miller, 1948; Wilkins et al., 1964; Lee et al., 1968). The second stage was a 3D simulation of the targeted military vehicle that uses the loading of the 2D simulation to assess the caused plastic strain and whether the explosion intruded the vehicle. Loverini and Robert (2023) did the same but used a different mass of TNT and distance from the military vehicle. Furthermore, Loverini (2023c) used OPENRADIOSS to model the detonation of a land mine based on the NATO regulations. They modelled the sand using a polynomial EoS and the air with an ideal gas EoS. They also placed a 50-percentile dummy inside to test the impact of the explosion on the dummy.

Besides AIREX, OPENRADIOSS is also ideal to investigate ballistics. Ferry et al. (2023) used OPENRADIOSS to simulate the impact of a bullet on a steel plate. They simulated only a quarter of the bullet and assumed the rest of the impact is symmetric. They find that the bullet is stopped by this plate of steel as demonstrated from experiments[1]. Loverini (2023a) used OPENRADIOSS to simulate the ballistic impact on a water tank. They used brick elements to simulate the aluminium tank, the Cole (1948) equation of state for water and the ideal gas law to model the air. Based on this the total damage to the tank was determined. Besides ballistics, explosive burning can also be simulated well. Loverini (2023d) simulated the propagation of explosive burning for an array of two different explosives and found that their result is as expected.

Fluid-structure interaction can also be simulated by OPENRADIOSS. Robert et al. (2023) simulated a bottle dropping to demonstrate the arbitrary Lagrangian-Eulerian

[1] The experiments stopped a higher velocity bullet.

(ALE) ability of OpenRadioss. They used thick-shell elements to model the bottle, an ideal gas EoS for air, and Cole (1948) EoS to model the water. These results agree with experiments of dropping water bottles. Robert and Loverini (2023) modelled a section of a boat using mirror symmetry to investigate the slamming of the boat using the same methods. Zheng et al. (2023) calculated the diving depth of pressure hulls before they will implode. They used cylindrical pressure hulls which were compared with experiments and gave good agreement.

Even more complicated simulations can be performed in OpenRadioss, notable, simulations of UNDEX. Loverini (2023b) simulated the hydroforming of spherical metal vessels. This is the construction of spherical metal containers with the help of explosives such that they become perfectly spherically. This again models the water using the Cole (1948) EoS and the air is simply modelled as a constant atmospheric load on the outside of the tank. Loverini (2023e) modelled an UNDEX event using a 2D simulation of 22.5 m by 11 m water reservoir. The overall behaviour of UNDEX in OpenRadioss is comparable to experiments. Only the peak pressure is slightly lower, but this is not surprising because the hydrodynamics of OpenRadioss is probably over smoothing. It is even able to correctly calculate the rarefaction wave that causes the cavitation in UNDEX events.

This wealth of different use cases shows that OpenRadioss is a widely used FEM package which can be used for a diverse set of different problems. OpenRadioss itself is not only a FEM package. It contains functionality to simulates fluids using ALE or grids with a large variation of different EoS. Most of the fluids can be simulated using the /MAT/HYDRO keyword. Specific EoS can be called by using a simpler keyword like /EOS/IDEAL-GAS for ideal gas EoS, /EOS/STIFF-GAS for the Cole (1948) EoS, /EOS/NASG for the Noble-Abel stiffened-gas EoS (Le Métayer and Saurel, 2016), and /EOS/JWL for the JWL EoS (Jones and Miller, 1948; Wilkins et al., 1964; Lee et al., 1968).

1.4 This report

In this report we focus on how OpenRadioss performs and do not make a direct comparison with LS-DYNA because TNO-NOS has experience with LS-DYNA and we expect that LS-DYNA works well for the different test cases shown in this report. For this report we used different versions of OpenRadioss between 01-09-2023 and 31-12-2023. The remainder of this report is structured as follows, Chapter 2 investigates the performance of shell and beam elements by comparing the numerical solution of a cantilever beam with the theoretical predictions. Chapter 3 investigates the performance of spring elements by comparing the numerical solution with the analytical or theoretical solutions. Chapter 4 investigates how to properly compile different parallelisation versions of the code and how to design your .bashrc in such a way that working with OpenRadioss can be done as efficiently as possible. Chapter 5 systematically investigates how to run OpenRadioss as efficiently as possible and how to change compiler settings to improve the performance. Chapter 6 summarises the advantages and disadvantages of OpenRadioss. Chapter 7 summarises the conclusions and recommendations of this report.

2 Cantilever beam

This chapter focuses on verifying shell and beam elements in OPENRADIOSS. This is achieved by designing a new benchmark based on the cantilever beam (see Fig. 2.1). A cantilever beam is a beam that is clamped on one side and is free on the other side of the beam. When a force is applied to the free side of the beam, the beam will start oscillating. Overall, the cantilever beam is an excellent benchmark because:

- Under general assumptions[1] the solutions of the cantilever beam can be calculated analytical. This means that the analytical solution can be directly compared to the numerical solution.
- The benchmark can be applied to both beam and shell elements with only a small change.
- The creation of finite element (FE) models requires only a rectangular grid without needing to refine regions[2].

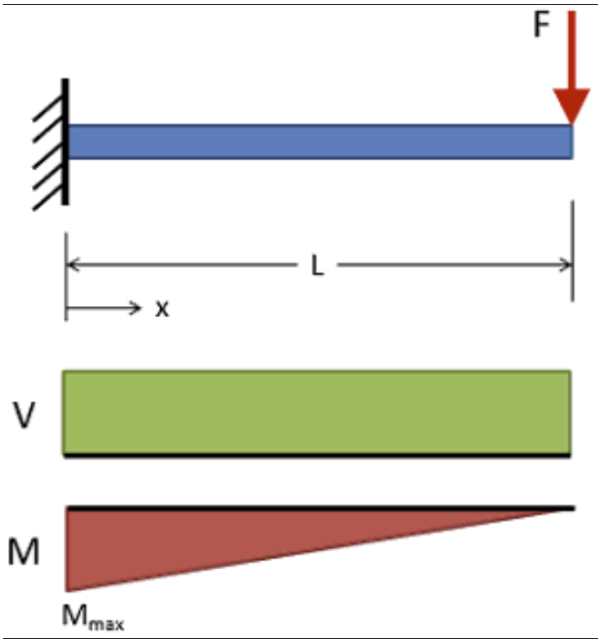


Figure 2.1 A schematic view of a cantilever beam. The cantilever beam is attached to the left side to a structure and has no degrees of freedom (DOF). On the right side all DOF are free and the cantilever beam can move here. For a known force F and a known material model the eigenfrequency and shape of the cantilever beam can be calculated analytically.

Furthermore, the cantilever beam simulations can be altered in a few ways to investigate different behaviour:

- Without damping: This is ideal to investigate the natural frequencies[3] and amplitudes of oscillations. The natural frequencies can be calculated directly from theory and compared with simulations.
- With Rayleigh damping: This is ideal to investigate the exact shape of the can-

[1] These are that the material is purely elastic, the thickness of the cantilever beam is not important, and the displacements are small.

[2] i.e. because the cantilever beam does not contain singularities. However, if smaller elements are used on the clamped side, this will result in convergence of the displacement to the exact solution with fewer elements

[3] Also known as the eigenfrequency

tilever beam when it is in equilibrium with the imposed force. The exact shape of the cantilever beam is also known from theory.
- The number of elements can be varied: The resolution of the cantilever beam can be varied by using a different number of elements to do a convergence test and see when the cantilever beam behaves well.
- The applied force can be rotated by 90°: This allows to also test the shell elements in the plane itself.
- The grid shapes can be modified: For example, triangular grids or even more arbitrary shapes to see how well the code converges when the grid is not rectangular (not done in this work).

The above text describes globally what the goal of this chapter is. The remainder of this chapter focuses on creating and testing this benchmark.

2.1 Analytical solution

To calculate the analytical solution of a cantilever beam, the following assumptions are made:

- It is assumed that the cantilever beam has a rectangular cuboidal shape (a hexahedron with only right angles[4]). Therefore, the length (L), width (b), and thickness (d) describe the shape of the cantilever beam.
- The material is assumed to be fully elastic, therefore, the material has only three free parameters namely, its initial mass density ρ, its Young's modulus E and its Poisson's ratio ν.

Based on these two assumptions, the second moment of area can be calculated as:

$$I = \iint_R y^2 \,\mathrm{d}x\,\mathrm{d}y = \int_{-b/2}^{b/2}\int_{-d/2}^{d/2} y^2 \,\mathrm{d}x\,\mathrm{d}y = \frac{bd^3}{12}. \tag{2.1}$$

2.1.1 Natural frequencies

One of the key properties of a system is its natural frequency (or eigenfrequency). In the case of a cantilever beam it is possible to calculate this analytically. Using the equation of motion (EoM) it is possible to calculate the natural frequencies of the cantilever beam by solving the EoM (Meirovitch and Wesley, 1967):

$$\frac{\partial^2}{\partial x^2}\left(EI(x)\frac{\partial^2 w(x,t)}{\partial x^2}\right) = \rho A(x)\frac{\partial^2 w(x,t)}{\partial t^2}, \tag{2.2}$$

where $A(x)$ is the area of the cross section and $w(x,t)$ is the displacement of the cantilever beam. Using separation of variables and assuming the time component is harmonic, the displacement can be written as

$$w(x,t) = w(x)\sin\left(\omega t + \phi\right), \tag{2.3}$$

where ω is the natural frequency and $w(x)$ is the vertical displacement that only depends on x. This means that the equation of motion reduces to

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}\left(EI(x)\frac{\mathrm{d}^2 w(x)}{\mathrm{d}x^2}\right) = \omega^2 \rho A(x) w(x). \tag{2.4}$$

Equation (2.4) can be rewritten under the assumption that $I(x) = I$ and $A(x) = A$ as:

$$\frac{\mathrm{d}^4 w(x)}{\mathrm{d}x^4} - \beta^4 w(x) = 0. \tag{2.5}$$

[4] Right angles are angles of exactly 90°.

where $\beta^4 = \frac{\omega^2 \rho A}{EI}$. Assuming that the displacement is given by:

$$w(x) = e^{\lambda x}, \tag{2.6}$$

means that the λ should satisfy $\lambda^4 = \beta^4$, therefore there are a total of four solutions namely:

$$\lambda_1 = \beta, \lambda_2 = -\beta, \lambda_3 = i\beta, \lambda_4 = -i\beta. \tag{2.7}$$

Rearranging these functions into standard functions gives the solution:

$$w(x) = A\cosh(\beta x) + B\sinh(\beta x) + C\cos(\beta x) + D\sin(\beta x). \tag{2.8}$$

The solution and its coefficients can be constrained by using the boundary conditions. For the cantilever beam the boundary conditions are:

$$w(0) = 0 \rightarrow A + C = 0, \tag{2.9}$$

$$\frac{\mathrm{d}w(0)}{\mathrm{d}x} = 0 \rightarrow B + D = 0, \tag{2.10}$$

$$\frac{\mathrm{d}^2w(0)}{\mathrm{d}x^2} = 0 \rightarrow A\beta^2\cosh(\beta l) + B\beta^2\sinh(\beta l) - C\beta^2\cos(\beta l) - D\beta^2\sin(\beta l) = 0, \tag{2.11}$$

$$\frac{\mathrm{d}^3w(0)}{\mathrm{d}x^3} = 0 \rightarrow A\beta^3\sinh(\beta l) + B\beta^3\cosh(\beta l) + C\beta^3\sin(\beta l) - D\beta^3\cos(\beta l) = 0. \tag{2.12}$$

The first two equations give $C = -A$ and $D = -B$. Using this to rewrite equation (2.11), a relation between A and B can be found as:

$$B = -A\frac{\cosh(\beta l) + \cos(\beta l)}{\sinh(\beta l) + \sin(\beta l)}. \tag{2.13}$$

In order to remove A or B we need to use equation (2.12) Equation (2.12) gives

$$A\sinh(\beta l) + B\cosh(\beta l) - A\sin(\beta l) + B\cos(\beta l) = 0, \tag{2.14}$$

because $\beta \neq 0$. The terms can be reordered as

$$A\left(\sinh(\beta l) - \sin(\beta l)\right) + B\left(\cosh(\beta l) + \cos(\beta l)\right) = 0, \tag{2.15}$$

Combining this with equation (2.13) gives

$$A\left(\sinh(\beta l) - \sin(\beta l) - \frac{(\cosh(\beta l) + \cos(\beta l))^2}{\sinh(\beta l) + \sin(\beta l)}\right) = 0. \tag{2.16}$$

A nontrivial solution is desired, so $A \neq 0$,

$$\sinh(\beta l)^2 - \sin(\beta l)^2 - (\cosh(\beta l) + \cos(\beta l))^2 = 0, \tag{2.17}$$

Because this expression contains many square terms, the equation can be reduced to (e.g. $\cos(a)^2 + \sin(a)^2 = 1$ and $\cosh(a)^2 - \sinh(a)^2 = 1$)

$$\cos(\beta l)\cosh(\beta l) + 1 = 0. \tag{2.18}$$

The solution of equation (2.18) need to be found with a root-finding algorithm[5], if this is done the first two solutions are

$$\beta_1 = \frac{0.6\pi}{l}, \tag{2.19}$$

$$\beta_2 = \frac{1.49\pi}{l}, \tag{2.20}$$

[5] By solving $\cos(\pi x)\cosh(\pi x) + 1 = 0$ in order to find the coefficients of the following solutions.

and for the later roots the solution is given by

$$\beta_n = \left(n - \frac{1}{2}\right)\frac{\pi}{l}. \tag{2.21}$$

for $n \geq 3$[6]. This means that the general solution of the natural frequency is given by

$$\omega_n = \sqrt{\frac{EI}{\rho A}}\beta_n^2, \tag{2.22}$$

which reduces to

$$\omega_n = \sqrt{\frac{EI}{\rho A}}\frac{\left(n - \frac{1}{2}\right)^2 \pi^2}{l^2}, \tag{2.23}$$

for $n \geq 3$. The benchmark in this chapter only requires the first five factors, these are given by $\beta_1^2/l^2 = 3.55$, $\beta_2^2/l^2 = 21.91$, $\beta_3^2/l^2 = 61.68$, $\beta_4^2/l^2 = 120.90$, $\beta_5^2/l^2 = 199.861$[7]. For a rectangular cross section the second moment of area is given by equation (2.1) and equation (2.23) can be rewritten to

$$\omega_n = \sqrt{\frac{Ed^2}{12\rho}}\frac{\left(n - \frac{1}{2}\right)^2 \pi^2}{l^2}. \tag{2.24}$$

this means that the natural frequency geometrically only depends on the length of the cantilever beam and its thickness d[8].

2.1.2 Displacement

Here the equilibrium displacement is calculated. The EoM for a cantilever beam with an imposed point force at $x = L$ is given by:

$$EI\frac{\mathrm{d}^4 w}{\mathrm{d}x^4} = F_0\delta(x - L). \tag{2.25}$$

where F_0 is the force on the cantilever beam and $\delta(x)$ is the Dirac delta function. To solve the EoMs, the equation is integrated four times[9]:

$$EI\frac{\mathrm{d}^3 w}{\mathrm{d}x^3} = F_0 + c_1, \tag{2.26}$$

$$EI\frac{\mathrm{d}^2 w}{\mathrm{d}x^2} = F_0 x + c_1 x + c_2, \tag{2.27}$$

$$EI\frac{\mathrm{d}w}{\mathrm{d}x} = \frac{1}{2}F_0 x^2 + \frac{1}{2}c_1 x^2 + c_2 x + c_3, \tag{2.28}$$

$$EIw = \frac{1}{6}F_0 x^3 + \frac{1}{6}c_1 x^3 + \frac{1}{2}c_2 x^2 + c_3 x + c_4. \tag{2.29}$$

The left side of the cantilever beam is clamped, and this directly means that the displacement ($w(x)$) and angle ($\mathrm{d}w/\mathrm{d}x$) are given by:

$$w(0) = 0 \rightarrow c_4 = 0, \tag{2.30}$$

$$\frac{\mathrm{d}w}{\mathrm{d}x}(0) = 0 \rightarrow c_3 = 0. \tag{2.31}$$

[6] We note that the solutions are rounded to two decimals.

[7] For β_1 and β_2 equations (2.19) and (2.20) are used while for $n \geq 3$ equation (2.21) is used.

[8] And the physical properties of the material E and ρ.

[9] Note that the integral of the Dirac delta function is given by $\int_{-\infty}^{\infty} \delta(x)\,dx = 1$.

The bending moment at L is given by:

$$EI\frac{\mathrm{d}^2w}{\mathrm{d}x^2}(L) = 0 \rightarrow F_0L + c_2 + c_1L = 0. \tag{2.32}$$

The shearing force at L is given by:

$$EI\frac{\mathrm{d}^3w}{\mathrm{d}x^3} = F_0 \rightarrow F_0 + c_1 = F_0 \rightarrow c_1 = 0. \tag{2.33}$$

The shearing force implies that the coefficients of the bending force result in:

$$c_2 = -F_0L. \tag{2.34}$$

Therefore, the displacement becomes:

$$w(x) = -\frac{1}{6}\frac{F_0}{EI}\left(3x^2L - x^3\right). \tag{2.35}$$

This solution and solution of other boundary conditions are often known as 'vergeet-mij-nietjes'. This means that the maximal displacement is given by:

$$w_{\max} = w(L) = -\frac{F_0}{3EI}L^3. \tag{2.36}$$

Based on the displacement the rotation angle can be calculated as:

$$\theta_x = \frac{\mathrm{d}w}{\mathrm{d}x} = \frac{1}{2}\frac{F_0}{EI}\left(x^2 - 2Lx\right). \tag{2.37}$$

Corresponding the bending moment is given by:

$$M(x) = EI\frac{\mathrm{d}^2w}{\mathrm{d}x^2} = F_0\left(x - L\right). \tag{2.38}$$

And the shearing force is given by:

$$V(x) = EI\frac{\mathrm{d}^3w}{\mathrm{d}x^3} = F_0. \tag{2.39}$$

These solutions can be used to test the performance of shell and beam elements.

A simulation without damping will on average reach the displacement and angle of equations (2.35) - (2.37). By including damping it is possible to test the convergence towards the analytical solution directly. This is what will be done later in this chapter.

2.2 Beam element formulations in Radioss

OpenRadioss uses the Timoshenko (1921, 1922) formulation for beam elements. The assumptions of the Timoshenko (1921, 1922) formulation are (Altair, 2022b):

- No cross-section deformation in the plane of the beam.
- No cross-section warping out of the plane of the beam.

In general, the equations describing the dynamics of a Timoshenko (1921, 1922) beam are given by:

$$\rho A\frac{\partial^2w}{\partial t^2} - q(x,t) = \frac{\partial}{\partial x}\left[\kappa AG\left(\frac{\partial w}{\partial x} - \varphi\right)\right] \tag{2.40}$$

$$\rho I\frac{\partial^2\varphi}{\partial t^2} = \frac{\partial}{\partial x}\left(EI\frac{\partial\varphi}{\partial x}\right) + \kappa AG\left(\frac{\partial w}{\partial x} - \varphi\right), \tag{2.41}$$

where G is the shear modulus, κ is the Timoshenko (1921, 1922) shear coefficient, $q(x,t)$ is the distributed load, and ϕ is the angle of rotation of the normal to the mid-surface of the beam. In OPENRADIOSS the user can change the beam formulation flag from $I_{\text{shear}} = 0$ to $I_{\text{shear}} = 1$, this allows the user to use Euler-Bernoulli beam theory which follows a simpler EoM given by (Reddy, 2017):

$$\frac{\partial^2}{\partial x^2}\left(EI\frac{\partial^2 w}{\partial x^2}\right) = -\rho A\frac{\partial^2 w}{\partial t^2} + q(x). \tag{2.42}$$

The main take away point from this formulation is that equations (2.40) and (2.41), but also equation (2.42) fully take into account all the terms of equation (2.2), the beam equation presented in § 2.1. Therefore, numerical convergence of the beam element formulations is expected because the analytical solution is one-on-one described by the numerical solution. The numerical solution is expected to find the correct solution if the time stepping of the simulation has a time step shorter than $\Delta t = aL/c_{\text{s}}$, where L is the beam length, c_{s} is the speed of sound and a is a constant of around unity (for details see Altair, 2022b).

2.3 Shell formulations

Plate theory is a mathematical model used to determine the deformation and corresponding stresses in thin plates that are subjected to forces and moments. Generally, there are two types of plate theory.

2.3.1 Kirchhoff-Love plate theory

The first type of plate theory is called Kirchhoff-Love (1888) (KL) plate theory. The three main assumptions of KL plate theory are (Reddy, 2006):

- Straight lines normal to the mid-surface remain straight after deformation.
- Straight lines normal to the mid-surface remain normal to the mid-surface after deformation.
- The thickness of the plate does not change during deformation.

The consequences of these assumptions are examined by considering the position vector of a point in an undeformed plate. The displacement of a point in a plate can be expressed as:

$$\mathbf{u}(\mathbf{x}) = u_1^0\mathbf{e}_1 + u_2^0\mathbf{e}_2 + w^0\mathbf{e}_3, \tag{2.43}$$

where $\mathbf{e}_i$ are the Cartesian unit vectors, u_α^0 is in-plane displacement and w^0 is the out-of-plane displacement in the x_3 direction. This means that the displacement is given by[10]:

$$u_\alpha(\mathbf{x}) = u_\alpha^0(x_1,x_2) - x_3\frac{\partial w^0}{\partial x_\alpha} \equiv u_\alpha^0 - x_3\partial_\alpha w^0, \alpha = 1,2 \tag{2.44}$$

$$u_3(\mathbf{x}) = w^0(x_1,x_2). \tag{2.45}$$

When the strains are infinitesimal the strain-displacement relations are given by:

$$\epsilon_{\alpha\beta} = \frac{1}{2}\left(\partial_\beta u_\alpha + \partial_\alpha u_\beta\right). \tag{2.46}$$

This implies that

$$\epsilon_{\alpha 3} = \frac{1}{2}\left(\partial_3 u_\alpha + \partial_\alpha u_3\right), \tag{2.47}$$

$$= -\partial_\alpha w^0 + \partial_\alpha w^0 = 0. \tag{2.48}$$

[10] The shortening of derivative operators is used, namely, $\frac{\partial}{\partial x_\alpha} = \partial_\alpha$

Similarly, for

$$\varepsilon_{33} = \partial_3 u_3(x_1, x_2) = 0. \tag{2.49}$$

This means that there are only 3 non-zero components given by:

$$\epsilon_{\alpha\beta} = \frac{1}{2}\left(\partial_\beta u^0_\alpha + \partial_\alpha u^0_\beta\right) - x_3 \partial_\alpha \partial_\beta w^0. \tag{2.50}$$

KL plate theory is only valid when the plate is thin enough such that the assumptions remain valid. This is the case when the ratio of the length and thickness does not exceed 20 (Altair, 2023). For ratios smaller than 20 it is important to use a plate theory which considers this.

2.3.2 Reissner-Uflyand-Mindlin plate theory

To overcome the issue of thicker plates, a second plate theory was developed called Reissner-Uflyand-Mindlin (RUM) plate theory. The main assumptions of RUM plate theory are:

- There is a linear variation of the displacement across the plate thickness.
- The thickness of the plate does not change during deformation.
- The normal stress through the thickness is ignored (i.e. the plane stress condition).

Most importantly, the assumptions of RUM plate theory imply that the angles that normal vectors make with the x_3 axis are no longer given simply by $\phi_\alpha = \partial_\alpha w^0$. Instead, the displacement vector is given by:

$$u_\alpha(\mathbf{x}) = u^0_\alpha(x_1, x_2) - x_3\phi_\alpha, \alpha = 1, 2 \tag{2.51}$$

$$u_3(\mathbf{x}) = w^0(x_1, x_2). \tag{2.52}$$

Based on this the strain-displacement relations are given by:

$$\epsilon_{\alpha 3} = \frac{1}{2}\left(\partial_3 u_\alpha + \partial_\alpha u_3\right), \tag{2.53}$$

$$= \frac{1}{2}\left(\partial_\alpha w^0 - \phi_\alpha\right). \tag{2.54}$$

Again $\epsilon_{33} = 0$ because u_3 does not depend on x_3. Lastly, the other components are given by:

$$\epsilon_{\alpha\beta} = \frac{1}{2}\left(\partial_\beta u^0_\alpha + \partial_\alpha u^0_\beta\right) - \frac{x_3}{2}\left(\partial_\beta \phi_\alpha + \partial_\alpha \phi_\beta\right). \tag{2.55}$$

Because the shear strain is constant across the thickness and it is known the shear stress should be parabolic it is required to incorporate a shear correction factor κ in equation (2.54). This means that equation (2.54) reduces to:

$$\epsilon_{\alpha 3} = \frac{\kappa}{2}\left(\partial_\alpha w^0 - \partial_\alpha \phi_\alpha\right). \tag{2.56}$$

Overall, in OPENRADIOSS the standard is to use RUM plate theory for shell elements (Altair, 2023), this means that OPENRADIOSS by default takes into account thick shells.

2.3.3 Shell formulations in OPENRADIOSS

This subsection will explain the different shell formulations that are available in OPENRADIOSS.

2.3.3.1 Belytschko-Tsay shell element

The first type of shell element is the Belytschko and Tsay (1983) shell element. In OPENRADIOSS these are called the classic Q4 elements. OPENRADIOSS has a total of four different hourglass penalisation methods to consider the Belytschko and Tsay (1983) shell element.

The Belytschko and Tsay (1983) shell elements use perturbation stabilisation for the hourglass energies. Hourglass modes are distortions of the mesh that have no strain energy. Hourglass modes only apply to 4 node shell elements. The left of Fig. 2.2 shows the 12 translational motions of the 4 node shell elements and the three hourglass modes. The right of Fig. 2.2 shows the 12 rotational modes of 4 node shell elements with the four rotational hourglass modes. Because hourglass modes do not have strain energy it is required to stabilise them to prevent the mesh deformations to take any random form. This is done using perturbation stabilisation.

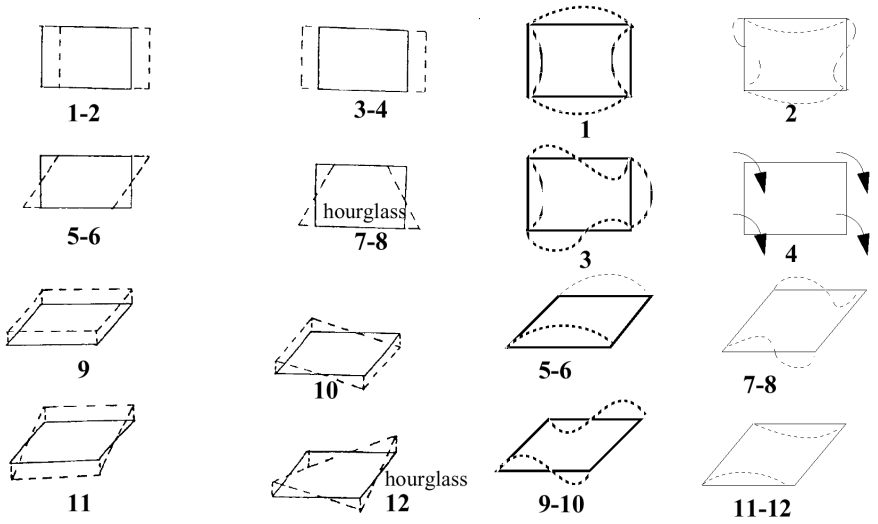


Figure 2.2 The different translational modes of 4 node shells with 7, 8 and 12 representing the three translational hourglass modes (left) and the different rotational modes of 4 node shells with 9-12 representing the four rotational hourglass modes of 4 node shells.

Perturbation stabilisation can be done in several different ways. Hourglass viscous forces relate the velocity of the nodes directly with a counteracting force, similar to viscous forces in a fluid. Hourglass stiffness forces work as a spring in which the stiffness force tries to counteract the force that wants to extend the nodes and therefore is directly related to the displacement. The hourglass viscous forces can also be applied to the moments of the shell. This means that there is a total of three hourglass forces that are required, namely an in-plane, out-of-plane and rotational hourglass force.

For the hourglass viscous force, the hourglass velocity rate is given by (Kosloff and Frazier, 1978):

$$\frac{\partial q_i}{\partial t} = \Gamma_\nu v_{i\nu} = v_{i1} - v_{i2} + v_{i3} - v_{i4}. \tag{2.57}$$

where Latin indices run over 1 and 2 and Greek indices run over 1 to 4. The hourglass normalised vector is given by

$$\Gamma = (1, -1, 1, -1). \tag{2.58}$$

The in-plane hourglass resisting forces at node ν and direction i are given by:

$$f_{i\nu}^{\mathrm{hgr}} = \frac{1}{4}\rho c d\sqrt{h_{\mathrm{m}}}\frac{A}{2}\frac{\partial q_i}{\partial t}\Gamma_\nu. \tag{2.59}$$

Where d is the element thickness, and h_{m} is the shell membrane hourglass coefficient. The out-of-plane hourglass resisting forces are given by:

$$f_{i\nu}^{\mathrm{hgr}} = \frac{1}{4}\rho c d^2\sqrt{\frac{h_{\mathrm{f}}}{10}}. \tag{2.60}$$

Where h_{f} is the shell out-of-plane hourglass coefficient (Altair, 2022b). The hourglass coefficients can be modified in the input of OPENRADIOSS and are by default set to $h_{\mathrm{m}} = 0.01$ and $h_{\mathrm{f}} = 0.01$ (Altair, 2022a).

Similarly hourglass elastic stiffness forces can be calculated, OPENRADIOSS follows the Flanagan and Belytschko (1981) formulation to do this. The hourglass resultant force is defined as:

$$f_{i\nu}^{\mathrm{hgr}} = f_i^{\mathrm{hgr}}\Gamma_\nu. \tag{2.61}$$

For the hourglass in-plane modes the hourglass energy is given by:

$$f_i^{\mathrm{hgr}}(t+\Delta t) = f_i^{\mathrm{hgr}}(t) + \frac{1}{8}h_{\mathrm{m}}Ed\frac{\partial q_i}{\partial t}\Delta t. \tag{2.62}$$

where t is the current time, Δt is the time step, and E is the Young's modulus. Similar for out-of-plane modes:

$$f_i^{\mathrm{hgr}}(t+\Delta t) = f_i^{\mathrm{hgr}}(t) + \frac{1}{40}h_{\mathrm{f}}Ed^3\frac{\partial q_i}{\partial t}\Delta t. \tag{2.63}$$

The last hourglass modes is the hourglass viscous moments, similar to equation (2.57) the angular velocity rate can be defined as:

$$\frac{\partial r_i}{\partial t} = \Gamma_\nu\omega_{i\nu} = \omega_{i1} - \omega_{i2} + \omega_{i3} + \omega_{i4}. \tag{2.64}$$

where ω is the angular velocity. The hourglass viscous moments are given by

$$m_{i\nu}^{\mathrm{hgr}} = \frac{1}{50}\sqrt{\frac{h_{\mathrm{r}}}{2}}\rho c A d^2\frac{\partial r_i}{\partial t}\Gamma_\nu, \tag{2.65}$$

where h_{r} is the shell rotation hourglass coefficient which by default is set to $h_{\mathrm{r}} = 0.01$ (Altair, 2022a).

The first penalisation method in OPENRADIOSS is $I_{\mathrm{shell}} = 1$ which corresponds to the Kosloff and Frazier (1978) and Flanagan and Belytschko (1981) formulation. This simply uses the default values described above of $h_{\mathrm{m}} = h_{\mathrm{f}} = h_{\mathrm{r}} = 0.01$. This method always calculates the velocity orthogonal to the physical velocity such that the hourglass velocity always remains orthogonal to the physical velocity.

The second penalisation method in OPENRADIOSS is $I_{\mathrm{shell}} = 2$. This method is identical to the previous method but it does not use the fact that the hourglass velocity remains orthogonal to the physical velocity (Altair, 2022a). In OPENRADIOSS this approach is called the Hallquist method.

The third penalisation method in OPENRADIOSS is $I_{\mathrm{shell}} = 3$. This method is called elastoplastic hourglass force. The elastoplastic hourglass method modifies the approach of Flanagan and Belytschko (1981) by imposing a minimum hourglass force. This means that equations (2.62) and (2.63) are modified to

$$f_i^{\mathrm{hgr}}(t+\Delta t) = \min\left(f_i^{\mathrm{hgr}}(t) + \frac{1}{8}h_{\mathrm{m}}Ed\frac{\partial q_i}{\partial t}\Delta t, \frac{1}{2}h_{\mathrm{m}}\sigma_{\mathrm{y}}d\sqrt{A}\right). \tag{2.66}$$

and

$$f_i^{\mathrm{hgr}}(t+\Delta t) = \min\left(f_i^{\mathrm{hgr}}(t) + \frac{1}{40}h_{\mathrm{f}}Ed^3\frac{\partial q_i}{\partial t}\Delta t, \frac{1}{4}h_{\mathrm{f}}\sigma_{\mathrm{y}}d^2\right). \tag{2.67}$$

where σ_{y} is the yield stress (Altair, 2022b). Also the default hourglass coefficients are different and are $h_{\mathrm{m}} = h_{\mathrm{r}} = 0.1$ and $h_{\mathrm{f}} = 0.01$ (Altair, 2022a), this means that there is a larger in-plane hourglass resistant force as compared to the other methods.

The fourth penalisation method in OPENRADIOSS is $I_{\mathrm{shell}} = 4$, this method specifically takes into account keeping the hourglass vectors orthogonal even in the case of warped elements. This penalisation method is the best in general for the Belytschko and Tsay (1983) shell element. $I_{\mathrm{shell}} = 4$ uses equations (2.57)-(2.65) but uses a slightly different hourglass vector (equation 2.58) that always remains orthogonal.

Note that the use of Belytschko and Tsay (1983) shell element is not without issue. One problem with Belytschko and Tsay (1983) shell elements is that they do not satisfy the Irons and Razzaque (1972) patch test (Haufe et al., 2013). The Irons and Razzaque (1972) patch test is a test of a couple of elements with solving a structure of a few elements for which the exact solution is known. The cantilever beam simulation is comparable to the Irons and Razzaque (1972) patch test but it is slightly more complicated by also including the dynamics. Based on this, convergence of the Belytschko and Tsay (1983) elements is not expected. A second problem with Belytschko and Tsay (1983) shell elements is that they show poor behaviour with irregular geometries. This means that they are unable to pass Irons and Razzaque (1972) patch tests with irregular geometries and are unable to pass the twisted beam test (e.g. the twisted beam tests of Macneal and Harder 1985 or Zupan and Saje 2004). A third issue with Belytschko and Tsay (1983) shell elements is that the hourglass coefficients are user inputs and are often taken as being constant, while in reality the hourglass coefficients are problem-dependent.

2.3.3.2 Fully-integrated QBAT shell element

Shell element formulation $I_{\mathrm{shell}} = 12$ is based on the Batoz and Dhatt (1990) Q4γ24 shell element. The Batoz and Dhatt (1990) shell element formulation has 4 nodes with each 5 DOF (RUM theory). Furthermore, the QBAT shell elements use a Cartesian shell approach where the middle surface is curved (instead of straight). Contrary to the other two shell elements, the QBAT shell element is fully integrated and uses four Gaussian quadrature points using 2×2 integration points. This means that the integration points for a rectangular shell element with size $L \times K$ are present at $\left(\frac{L}{2}\left(1-\frac{1}{\sqrt{3}}\right), \frac{K}{2}\left(1-\frac{1}{\sqrt{3}}\right)\right)$, $\left(\frac{L}{2}\left(1-\frac{1}{\sqrt{3}}\right), \left(\frac{K}{2}\left(1+\frac{1}{\sqrt{3}}\right)\right)\right)$, $\left(\frac{L}{2}\left(1+\frac{1}{\sqrt{3}}\right), \frac{K}{2}\left(1-\frac{1}{\sqrt{3}}\right)\right)$, and $\left(\frac{L}{2}\left(1+\frac{1}{\sqrt{3}}\right), \left(\frac{K}{2}\left(1+\frac{1}{\sqrt{3}}\right)\right)\right)$. Furthermore, a reduced in-plane integration for shear aims at preventing the QBAT from shear locking. Similar to the Belytschko and Tsay (1983) shell elements an hourglass force needs to be imposed, but contrary, the hourglass force is physically motivated based on the Belytschko et al. (1984) formulation. In practice OPENRADIOSS uses an updated formulation based on the Belytschko and Leviathan (1994) and Belytschko et al. (1984) formulation. Because the hourglass energy is modelled using a physical model this method does not output any hourglass energies (Altair, 2023).

Of all the shell element types QBAT is the most expensive because it is a full integration scheme. Due to this it is a scheme that is not often used in simulations that use explicit time integration. Rather it is more commonly used in simulations that use implicit time integration, still, this scheme can be used in simulations with explicit time integration but might produce locking (e.g. Zeng and Combescure, 1998).

2.3.3.3 Reduced-integration QEPH shell element

The last shell formulation is $I_{\text{shell}} = 24$, QEPH shell elements are cheaper than QBAT shell elements because this shell formulation just requires one number of integration point in the shell element because it is a reduced integration scheme with one instead of four integration points. Similar to QBAT shell elements the hourglass energy is modelled using a physical model using an updated Belytschko and Leviathan (1994) and Belytschko et al. (1984) formulation, which is explained in Zeng and Combescure (1998). Zeng and Combescure (1998) shell elements give nearly perfect agreement with the fully integrated Batoz and Dhatt (1990) Q4γ24 shell elements for linear problems. Compared to Batoz and Dhatt (1990) Q4γ24 shell elements, Zeng and Combescure (1998) shell elements only perform worse in the case of a significantly coarser mesh for linear problems. The main advantage of Zeng and Combescure (1998) shell elements is that compared to QBAT or Batoz and Dhatt (1990) Q4γ24 shell elements the number of computations is a factor of 4 to 5 lower. This means that Zeng and Combescure (1998) shell elements are only a factor of 20 per cent slower than the Belytschko and Tsay (1983) shell elements.

It is noted that Zeng and Combescure (1998) shell elements have not been tested extensively for nonlinear materials, for example anisotropic damaged materials. This means that users should be careful when using Zeng and Combescure (1998) shell elements for problems that involve nonlinear materials. Furthermore, LS-DYNA does not have the option to run with Zeng and Combescure (1998) elements so no experience with these shell elements exists in TNO-NOS.

2.3.4 Thickness integration

This subsection explains how thickness integration is done for shell elements in OPENRADIOSS. The thickness integration is based on Gaussian quadrature rules (Gauss, 1815). In the parameter file the user specifies the amount of integration points n, where n can range from 1 to 9. To integrate over the thickness of the shell the default numerical integration approaches for functions of the shape

$$\int_{-1}^{1} f(x)\mathrm{d}x. \tag{2.68}$$

are used. By default OPENRADIOSS uses Gauss-Lobatto quadrature (see page 888 of Abramowitz and Stegun, 1965), this means that two of the integration points are taken at the edge of the thickness. This means that the integral is calculated as:

$$\int_{-1}^{1} f(x)\mathrm{d}x = w_1 f(-1) + w_n f(1) + \sum_{i=2}^{n-1} w_i f(x_i). \tag{2.69}$$

where the endpoints have a weight of

$$w_{1,n} = \frac{2}{n(n-1)}, \tag{2.70}$$

and the other weights are given by simply the Gaussian Quadrature weights of:

$$w_i = -\frac{2n}{(1-x_i^2)P''_{n-1}(x_i)P'_m(x_i)} = \frac{2}{n(n-1)\left(P_{n-1}(x_i)\right)^2}. \tag{2.71}$$

Where x_i is the i-th root of P_n, where P_n are the Legendre polynomials. OPENRADIOSS has several types of shell elements, not to be confused with the I_{shell} element

that can be selected in the standard shell element type (TYPE1). OPENRADIOSS has several different shell element types that mainly differ in the way vertical integration is done and if the material consists out of different materials. The simple shell element types (TYPE1) always use Gauss-Lobatto quadrature, and assume there is just one type of material in the shell. In OPENRADIOSS it is possible to use Gauss-Legendre quadrature (Gauss, 1815), however, this is not possible to do with the simple shell type. To use Gauss-Legendre quadrature element type 10 or 11 (TYPE10 or TYPE11) need to be used. Type 10 and 11 are the composite shell and sandwich shell respectively. In the case of Gauss-Legendre the integral is calculated as:

$$\int_{-1}^{1} f(x)\mathrm{d}x = \sum_{i=1}^{n} w_i f(x_i). \tag{2.72}$$

where the coefficients are given by:

$$w_i = \frac{2}{(1-x_i)^2 \left(P_n'(x_i)\right)^2} \tag{2.73}$$

So, what is the accuracy of both elements? In general Gaussian quadrature can exactly integrate a polynomial of order $2n-1$. This means that for a correct calculation of the cantilever beam having only 2 thickness integration points would be enough to correctly calculate the displacement of the cantilever beam.

2.4 Constructing the cantilever beam simulations

To effectively run simulations of cantilever beams (clamped-free) and similar structural elements with different boundary conditions a python module was designed to create simple Radioss starter files. The details of the program can be found in Appendix C. A short explanation will be given here. The python module can be run by executing

```
> python3 create_rectangular_test_grid.py parameter_file.yml >
    output_radioss_file_0000.rad
```

Here CREATE_RECTANGULAR_TEST_GRID.PY is the python module, PARAMETER_FILE.YML is the parameter file and OUTPUT_RADIOSS_FILE_0000.RAD is the resulting file for Radioss. The parameter file for the simulation to create consists out of different subsections for each component of the model with separate variables that can be easily set by the user, the input looks like:

```
BeamDimension:
        Length:      40.0
        Width:       10.0
        Thickness:   2.5
        CoordinateSystemZeroPoint: [0.,0.,0.]

AppliedForce:
        TotalForce:    0.1
        TypeOfForce:   "line force"

NumericalResolution:
        TypeOfElement:      "beam"
        ShellElementSize:   5.0
        BeamElementSize:    5.0
```

```
        BeamMembraneDamping:          0.0
        BeamFlexuralDamping:          0.01
        VerticalIntegrationPoints:    5
        ShellFourFormulation:         24
        ShellNumericalDamping:        0.015
        BeamFormulationFlag:          0
        SmallStrainOptionFlag:        4

MaterialProperties:
        LawNumber:          1
        Density:            7.8e-6
        YoungsModulus:      210.0
        PoissonRatio:       0.3
        YieldStress:        0.3
        UltimateTensileEngineeringStress: 0.686
        EngineeringStrainAtUTS: 0.129
        StrainRateCoefficient:  0.0
        FailureModel: False
        InputTypeFlag: 1

InternalUnitSystem:
        UnitMass:           "kg"
        UnitLength:         "mm"
        UnitTime:           "ms"

MetaData:
        RunName:            "Cantilever beam"
        Author:             "Folkert Nobels"

Damping:
        UseDamping:     True
        RayleighMassDamping:   4.0
        RayleighStiffnessDamping: 1.0
        StartTime:          0.0
        EndTime:            1e30
```

The yield stress and ultimate tensile engineering stress were taken from another example. In the linear analysis we do not use the yield stress and the ultimate tensile engineering stress.

2.5 Cantilever beam simulation

Fixed dimensions are taken for the cantilever beam, the aim is to have shell and beam element models with the same period, so the length and the thickness are identical for the shell and beam element model simulations. Table 2.1 summarises the physical parameters of the cantilever beams. For both the shell and beam elements, simulations with different numerical resolutions are constructed. The shell elements are constrained to have an initial squared shape, an additional requirement is that there are nodes in the middle of the cantilever beam. This means, for the dimensions in Table 2.1, that the lowest resolution simulation for shell elements has $8 \times 2 = 16$ shell elements. For beam elements there is no such constraint because beam elements are one dimensional and the simulation therefore can be done with just one

beam elements. For shell elements six simulation variations are performed with shell element sizes of:

$$\Delta x_{\text{shell}} = \frac{500 \text{ mm}}{2^n}, n = 0, 1, 2, 3, 4, 5. \tag{2.74}$$

For beam elements 3 more resolution variations are performed and the beam element size is given by:

$$\Delta x_{\text{beam}} = \frac{4000 \text{ mm}}{2^n}, n = 0, 1, 2, 3, 4, 5, 6, 7, 8. \tag{2.75}$$

The width of the shell and beam elements are all set to 8 mm. The highest resolution simulation has a resolution size of 15.625 mm, this means that the thickness over element size is at most 0.512. This means that the use of shell and beam elements is appropriate. Also, for shell element we used a width of 1 m because the additional constraint is that the shell element is rectangular. Furthermore, to not complicate the story of beam elements we also use a width of 8 mm such that it is equal to the thickness. Fig. 2.3 shows a snapshot of the cantilever beams at different spatial resolutions, it shows the shell elements (black lines) and the vertical displacement at the time of the snapshot. To be able to compare the displacement between the shell and beam elements, the force put on the free side of the cantilever beam is taken such that the maximal displacement is expected to be around 40 mm. A displacement of 40 mm is taken because this corresponds to a 1 per cent vertical displacement which should not cause significant thickness or bending effects such that comparison with the theoretical predictions in § 2.1.1 and 2.1.2 can be done. Due to a relatively slightly larger loading for the shell elements, which, in hindsight, should have been 1.6×10^{-2} kg mm ms^{-2}, there is a slight difference in displacement between shell and beam elements.

Table 2.1 Physical dimensions of the cantilever beam for shell and beam elements.

	shell elements	beam elements
length (L)	4 m	4 m
width (b)	1 m	8 mm
thickness (D)	8 mm	8 mm
force (F)	1.68×10^{-2} kg mm ms^{-2}	1.28×10^{-4} kg mm ms^{-2}
density (ρ)	7.8×10^{-6} kg mm^{-3}	7.8×10^{-6} kg mm^{-3}
Young's modulus	210 GPa	210 GPa

Two cases are considered for the cantilever beam simulations:

1. No damping: This allows verification that the solution is stable and there is no artificial damping present in the simulation. This is crucial for determining the natural frequencies of the solution and the comparison with § 2.1.1. Furthermore, this allows verification of the numerical integration scheme to check how stable the solution is and no energy is lost or gained due to numerical errors.
2. Rayleigh damping: This will produce a stable static solution which can be compared directly with the predicted displacement in § 2.1.2

The important question is: How good is the convergence expected to be? And is there a difference between beam and shell elements?

The beam elements are modelled with the Timoshenko beam theory (equations 2.40 and 2.41), the cantilever beam can be modelled with equation (2.2). Equation (2.2) is a specific case of equations (2.40) and (2.41), this means that very good convergence is expected for the beam elements.

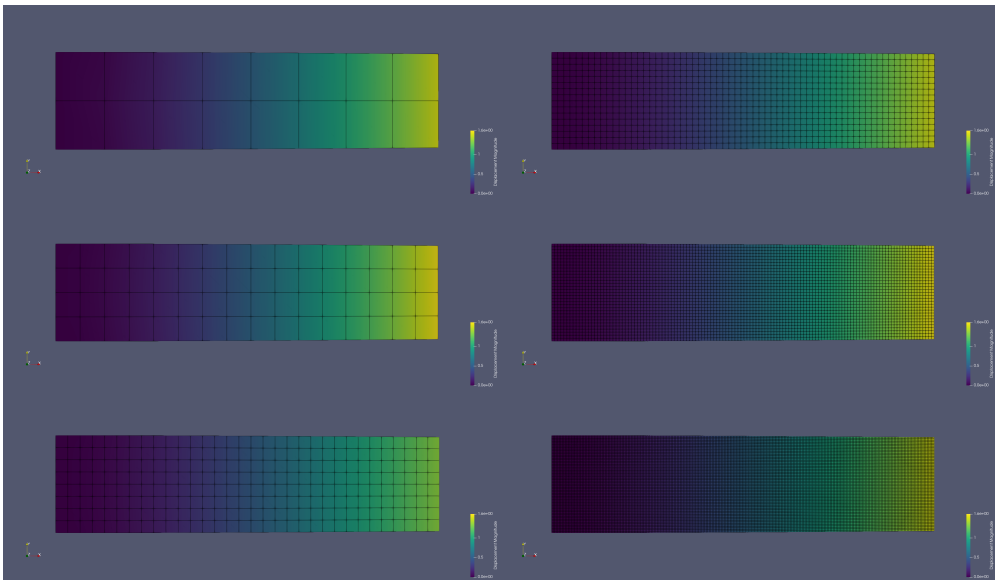


Figure 2.3 A time snapshot impression of the cantilever beam from top view with different numerical resolutions. The left shows from top to bottom the simulations with numerical resolutions of 500 mm, 250 mm and 125 mm, the right shows from top to bottom the numerical resolutions of 62.5 mm, 31.25 mm and 15.6125 mm. The colour bar indicates the vertical displacement (same in all plots) and the black squares show the edges of the shell elements.

For the shell elements the situation is different. Firstly, shell elements do not solve the beam theory equations, they rather extend the beam theory equations and generalise this to plates. Because of the assumptions of plate theory, the strains with z components become 'artificial' and are more numerical corrections than actual strains in the components $\epsilon_{\alpha 3}$ and ϵ_{33}. Often this also means that ϵ_{33} becomes zero[11].

[11] ϵ_{33} becomes zero for KL and RUM plate theory, other plate theories do not have this equal to zero like plate theories with stretchable cross-sections or non-straight cross-sections.

2.5.1 Vertical displacement

Here quantitatively the result of the vertical displacement is compared with the theoretical predicted result for the cantilever beam simulations. A comparison is made between shell and beam elements.

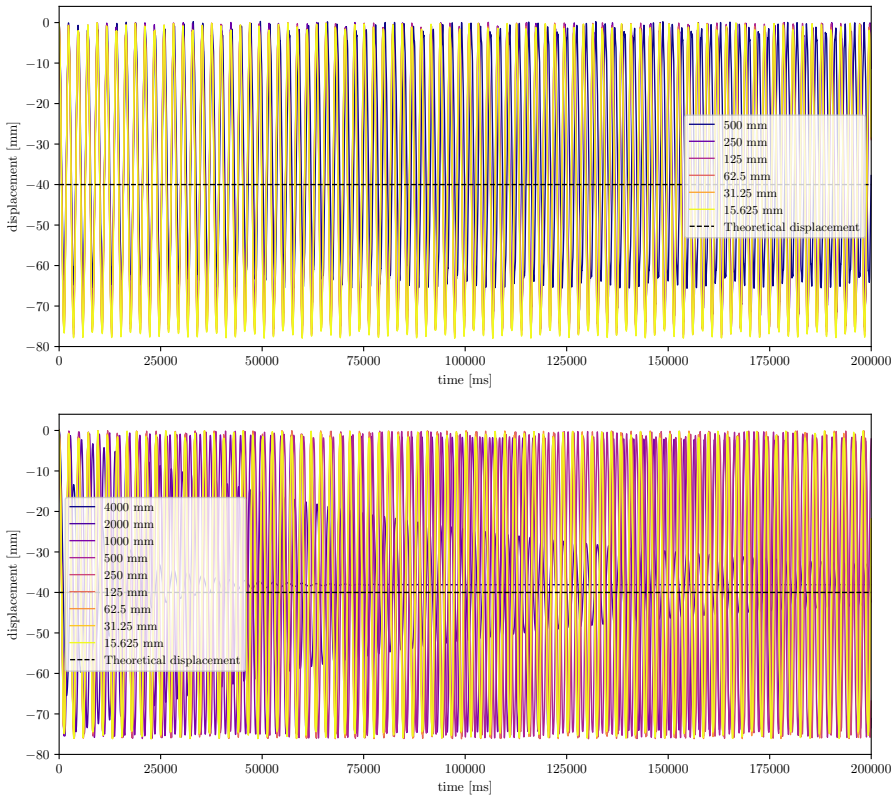


Figure 2.4 Time evolution of the cantilever beam with the Zeng and Combescure (1998) shell elements (top) and Timoshenko (1921, 1922) beam elements (bottom) for different numerical resolutions (different colours). The displacement of the edge of the cantilever beam is shown at a distance of 4000 mm. A comparison is made with the theoretical mean displacement (black dashed line). Simulations with low resolutions are unable to get the correct mean displacement when shell elements are used. Shell elements with low resolution have the wrong downwards amplitude while the upper amplitude is well converged. Beam amplitudes initially have the correct amplitude but beam elements with few resolution elements quickly artificially damp.

Fig. 2.4 shows the time evolution of the cantilever beam for shell and beam elements for around 100 cycles. The lower resolution shell elements do not have the correct mean displacement, this is mainly because the oscillation does not extend to low enough displacements. Only at a resolution of 125 mm, the solution becomes well converged. The behaviour of beam elements is different, beam elements have the correct upper and lower displacement. The lowest resolution simulations have the problem that they artificially damp quickly within 200 seconds. The convergence of beam elements is good when the cantilever beam is resolved by 4 beam elements (resolution of 1000 mm).

Fig. 2.4 shows the vertical displacement and from this the natural frequency of the oscillation can be determined. The most straightforward way of determining the natural frequencies is calculating the Fourier transform of the displacement. The Fourier transform calculates the contributions of different frequencies ν to the input signal, in our case the vertical displacement. Ideally, the mean displacement is subtracted to prevent an artificially peak around $\nu = 0$ which could reduce signals at the first natural

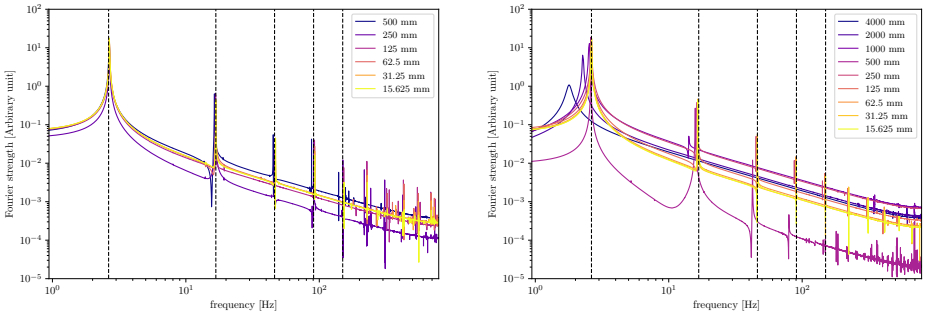


Figure 2.5 Absolute value of the Fourier transfer of the displacement minus the mean displacement for Zeng and Combescure (1998) shell elements (left) and Timoshenko (1921, 1922) beam elements (right). Different colours show the different numerical resolutions. The vertical black dashed lines indicate natural frequencies of the cantilever beam. When the numerical resolution is increased the convergence of the natural frequencies improves. Beyond the fifth natural frequency the amount of noise increases significantly.

frequency. Fig. 2.5 shows the absolute value of the Fourier transform and shows that the natural frequencies (black dashed lines) converge well with resolution. For shell elements all the natural frequencies are well converged even for the lowest resolution simulations. For beam elements, the natural frequencies are converged when the cantilever beam is resolved by four beam elements. This is partly as expected, we see that for one beam element only one natural frequency is produced, for two beam elements there are only two natural frequencies. This is because the model with N beam elements has only N degrees of freedom and can describe N number of natural frequencies at most. Overall, the agreement between both is good and indicates that the solution is converged when the cantilever beam is resolved by 4 or 8 elements in the length.

Fig. 2.5 also shows the slope of the Fourier transform. The slope can be determined to be $\mathscr{F} \propto \nu^{-0.4}$[12]. An interesting property of a Fourier transform is the energy power spectrum. The energy power spectrum is a measure of energy of the different Fourier transformation modes and can be calculated as $E = \mathscr{F}^2$. In Fig. 2.5 this corresponds to slope of the energy power spectrum of $E \propto \nu^{-0.8}$. The energy of the energy power spectrum does not matter much because it is directly related to the displacement (i.e. a constant that can be taken outside of the Fourier transform). Contrary, the slope tells how much the energy is divided between the different modes in the system. For Fig. 2.5 the slope of the energy power spectrum follows almost pink noise (pink noise has an energy spectrum of $E \propto \nu^{-1}$) besides the five peaks at the natural frequencies. Pink noise is one of the most common energy power spectra found in nature (e.g. tides, heart beats, neurons, black holes, etc.) Because it is so common this implies it is understood quite well and implies that the energy contribution of higher frequencies half when doubling the frequencies. However, in the case that the energy power spectrum follows pink noise, this means that the total energy of the cantilever beam does not converge with higher energy because:

$$E_{\text{total}} \propto \int_{\nu_{\text{min}}}^{\nu_{\text{max}}} \nu^{-1} \mathrm{d}\nu \propto \ln\left(\frac{\nu_{\text{max}}}{\nu_{\text{min}}}\right), \tag{2.76}$$

where ν_{max} and ν_{min} are the maximum and minimum frequency in the simulation. in the limit that $\nu_{\text{max}} \to \infty$ this gives a diverging result, implying that if the numerical

[12] Here the symbol $\propto$ shows that the two quantities are proportional to each other.

resolution is increased the total energy keeps increasing. The maximal frequency depends directly on the numerical resolution because simulations with high resolutions have shorter time steps. For the shell element that we considered, the total energy of the energy power spectrum converges even less slow as

$$E_{\text{total}} \propto \int_{\nu_{\text{min}}}^{\nu_{\text{max}}} \nu^{-0.8} \mathrm{d}\nu \propto \nu_{\text{max}}^{1/5} - \nu_{\text{min}}^{1/5}, \tag{2.77}$$

implying that when the resolution is increased the total energy increases with the fifth root of the maximum frequency. Overall, this result implies that convergence is not quaranteed for these shell elements. Specifically, this means that the shell elements show ultraviolet divergence (i.e. the solution diverges at the highest energies or frequencies).

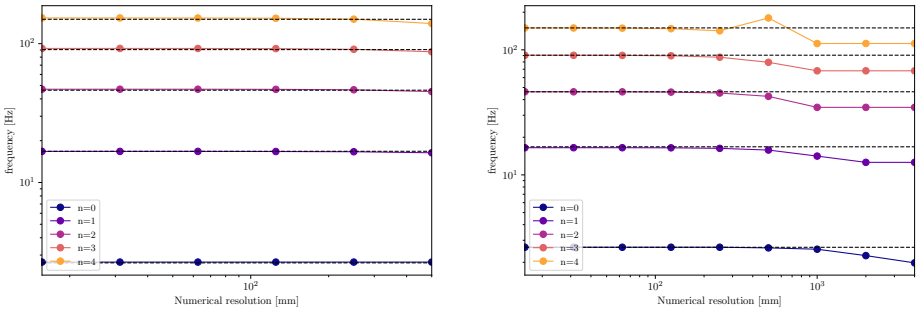


Figure 2.6 Convergence of the natural frequencies (different colours) for different numerical resolutions (x-axis) for Zeng and Combescure (1998) shell (left) and Timoshenko (1921, 1922) beam elements (right). The $n = 0$ natural frequency is converged for all shell element sizes investigated (i.e. from 500 mm onwards), the convergence of the beam elements starts at a spatial resolution of 500 mm, in excellent agreement between the two different element formulations. The convergence of the higher natural frequencies ($n > 0$) is well converged for spatial resolution of 250 mm and 125 mm for the shell and beam elements.

Fig. 2.6 shows the convergence of the five lowest natural frequencies in a more quantitative comparison. The convergence is excellent for both elements at a resolution of 125 mm for all natural frequencies. This implies that over the length a total of 32 elements are required. Shell elements already converge well for a lower resolution of 250 mm, so a total of 16 elements over the length are required. Lastly, the $n = 0$ natural frequency converges for the beam elements at a factor 2 or 4 lower resolution than the higher natural frequencies. The convergence study for the shell elements shows that most results are already converged for the coarsest model, making it impossible to judge for the shell elements how convergence of the $n = 0$ mode relates to the other modes.

To investigate the exact mean displacement behaviour more quantitatively a fit to the mean displacement curve is made with the curve of a damped mass-spring system:

$$z(t) = A \sin(\omega t + \phi) \exp\left(-\frac{t}{t_{\text{decay}}}\right). \tag{2.78}$$

where A is the amplitude, ω is the $n = 0$ natural frequency, ϕ is the phase offset, and t_{decay} is the time decay. As the system is only lightly damped, it is expected that ω is close to ω_0, $\phi \approx \pi/2$, $A \approx \langle z \rangle$ and t_{decay} is very large. This equation is nonlinear in the parameters that are required to be fitted. There are a few issues with this equation, firstly ω and ϕ are partly degenerate, secondly, both parameters have

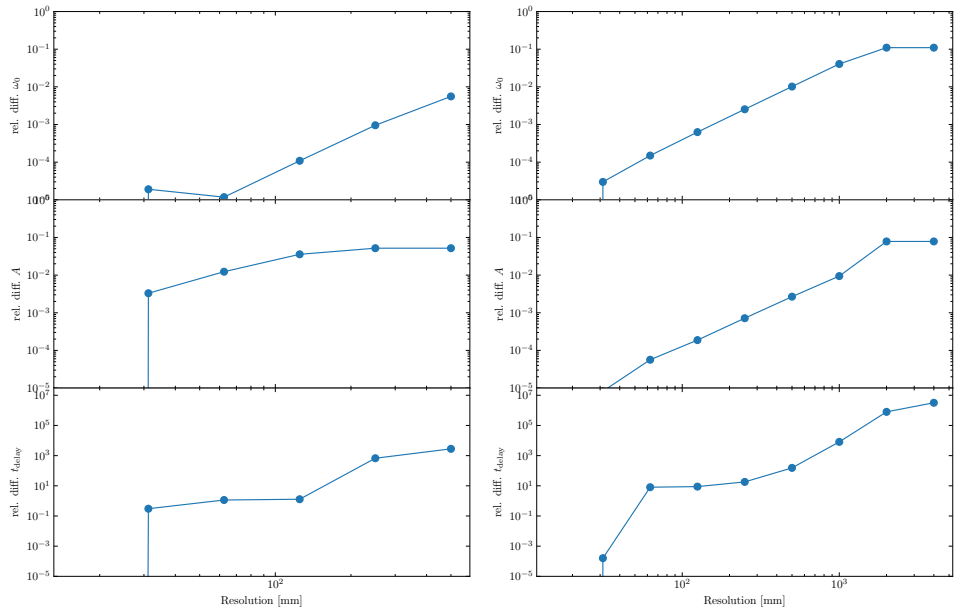


Figure 2.7 Convergence of the fitted values as a function of spatial resolution (x-axis) for shell (left) and beam elements (right). The relative difference of the frequency (top) converges very well and at the highest resolutions is more accurate than 10^{-4}. The amplitude (middle) is well converging for the beam elements while for the shell elements the relative difference does not get below 10^{-3}. The delay time (bottom) also converges well for higher resolution but has high values because the delay time is expected to be very long.

degeneracies at $\omega + 2\pi n$, $n \in \mathbb{Z}$. This implies that using the Levenberg-Marquardt algorithm would most likely get stuck in local minima and a fitting procedure is required that uses bounds on both ω and ϕ. Additionally, $A > 0$ and $t_{\mathrm{decay}} > 0$ so additional boundary conditions can be imposed. The Branch et al. (1999) trust region reflective algorithm is used, this method converges well. Fig. 2.7 shows the convergence of the fitted parameters A, ω and t_{decay} as a function of spatial resolution. The frequency is quite well converged and is converged better for shell elements than beam elements, in-line with the findings in Fig. 2.6. The amplitude is less well converged for the simulation with shell elements while for beam elements the convergence is as good as for the frequency. Good convergence is not expected for the decay time because it can vary over many orders of magnitude. However, the decay time shows some convergence, the convergence implies that models with lower resolutions are more artificially damped than the highest resolution simulations.

Fig. 2.8 shows the mean displacement of the cantilever beam with the 16$^{\text{th}}$ and 84$^{\text{th}}$ percentile of the displacement indicated by the shaded colours[13]. At higher resolutions the simulations converge towards the equilibrium displacement. For the Zeng and Combescure (1998) shell elements the 84$^{\text{th}}$ percentile converges very rapidly (In-line with the findings of Fig. 2.4). However, the 16$^{\text{th}}$ percentile converges much slower, it requires a spatial resolution of at least 62.5 mm for convergence. This can be seen in the figure that above the line there is an uniform brown shaded region, while below there is a brown shaded region with more yellow colours below it that correspond to higher resolution simulations. The situation is different for the Timoshenko (1921, 1922) beam elements, the mean value converges rapidly even for the

[13] The 16$^{\text{th}}$ and 84$^{\text{th}}$ percentile correspond to $\pm 1\sigma$ (standard deviation) for a normal distribution. For an unknown distribution it is better to use percentiles instead of standard deviation because it applies to more general distributions. Information about percentiles can be found here

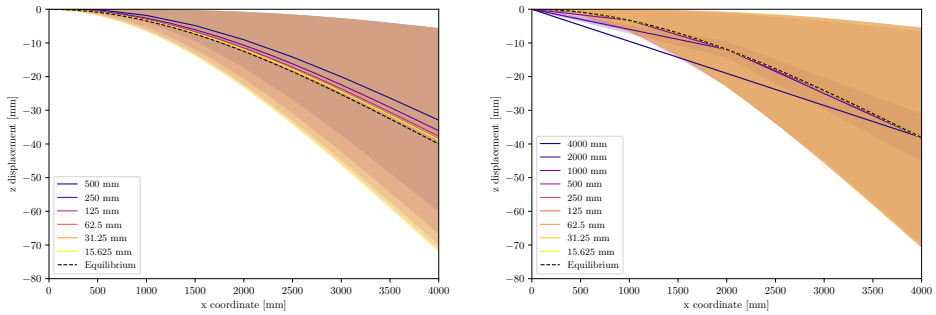


Figure 2.8 Comparison of the mean displacement (continuous lines) and the 16th and 84th percentile (shaded regions) for simulations of different spatial resolutions (different colours). For shell elements the Zeng and Combescure (1998) formulation is used. Shell elements converge from a spatial resolution of 62.5 mm (i.e. 64 resolution elements in the length), lower resolutions do not converge in the mean displacement and the 16th percentile vertical displacement (the lighter shaded regions). For Timoshenko (1921, 1922) beam elements the convergence is very good, the mean displacement is converged at 1000 mm (i.e. 4 resolution elements).

lowest resolutions. The two lowest resolution simulations do not predict the correct amplitude because they are artificially damped but predict the correct mean displacement. For simulations with only 4 elements (i.e. 1000 mm) the convergence of the 16th and 84th is already excellent. We think that the difference might be related to the high frequencies of Fig. 2.6. At the highest frequencies (higher than fifth natural frequency) there are way more peaks for the shell elements. These peaks are likely more present for shell elements because the FEM model with shell elements has more DOF. Due to the higher number of DOF there are also more natural frequencies that can perturb the result.

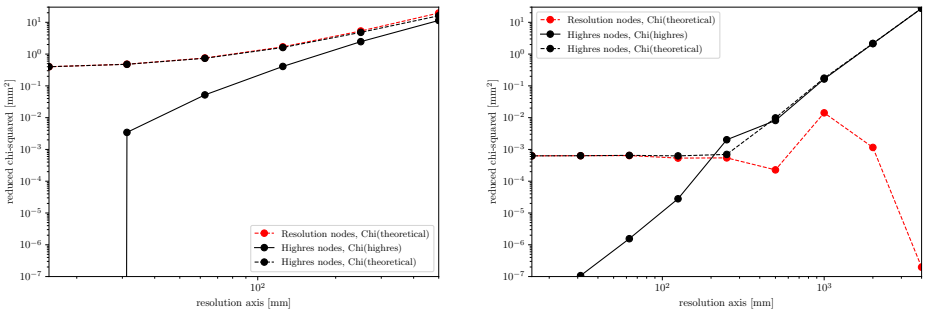


Figure 2.9 Reduced χ-squared of the mean displacement of Fig. 2.8 using three different approaches for Zeng and Combescure (1998) shell elements and Timoshenko (1921, 1922) beam elements. The reduced chi-squared is calculated based on the displacement of the cantilever beam. Either the difference is compared to the theoretical curve (theoretical) or the highest resolution simulation (highres). Additionally, the resolution nodes or highres nodes are used as a measurement position for the reduced chi-squared. The Timoshenko (1921, 1922) beam elements show excellent convergence with respect to their highres simulations and good convergence with respect to the theoretical solution which quickly converges towards 10^{-3}. Zeng and Combescure (1998) shell elements show not so good convergence compared to the theoretical curves and show convergence a bit better than 10^{-2} with respect to the highest resolution theoretical simulation.

Fig. 2.9 shows the χ-squared difference between the mean displacement and the highest resolution simulation. Timoshenko (1921, 1922) beam elements show excellent convergence towards their highest resolution simulation. The convergence with the theory is around 10^{-3}. Clearly, this is not excellent, but it is not expected that

this is perfect because the comparison with the theoretical solution does assume that there is no change in the x-position of the cantilever beam, but because it deviates slightly it is expected that the theoretical displacement will be slightly off. Given that the deviation is 1 per cent of the cantilever beam a relative offset of slightly less than one per mille is therefore not surprising. Lastly, the Timoshenko (1921, 1922) beam elements clearly converge rapidly with resolution $\chi \propto \Delta x^4$, therefore increasing the improvement with resolution is Δx^2.

For Zeng and Combescure (1998) shell elements the convergence with the highest resolution simulation is not that good, the model only converges towards a reduced χ-squared of $\sim 3 \times 10^{-2}$. The convergence with respect towards the theoretical solution deviates around five per cent because the Zeng and Combescure (1998) shell elements are too stiff. The convergence with respect to the highest resolution simulation shows that $\chi \propto \Delta x^{2.3}$, therefore the improvement with resolution is only $\sim \Delta x^{1.17}$ and this shell element model does not converge that fast.

2.5.2 Accuracy of the solution for different time step sizes

Like most codes that solve partial-differential equations (PDEs) that depend on time, OPENRADIOSS uses the Courant et al. (1928) (CFL) condition as a time step criterion to constrain the maximal allowed time step. The CFL condition requires the time step Δt to satisfy (CFL)

$$\Delta t \leq \frac{l_{\mathrm{c}}}{c_{\mathrm{s}}}. \tag{2.79}$$

where l_{c} is the characteristic element/resolution length and c_{s} is the speed of sound. This CFL condition corresponds to the time step that a sound wave travels through a single resolution element without skipping a single resolution element. Because this corresponds to the minimum time required for a sound wave to travel through a resolution element, in general, a slightly stricter time step criterion is imposed, and it is common to use

$$\Delta t \leq C_{\mathrm{CFL}} \frac{l_{\mathrm{c}}}{c_{\mathrm{s}}}, \tag{2.80}$$

where C_{CFL} is the CFL constant. The CFL condition is required to be satisfied for a stable solution of a PDE, this means that the CFL condition applies both on fluids and solids. In general, for SPH and ALE the CFL condition is required to be smaller simply because inside a SPH and ALE kernel there could be multiple resolution elements, implying that l_{c} is effectively smaller than the full width at half maximum (FWHM) of the kernel width which is often denoted by h. In practice in SPH $C_{\mathrm{CFL}} = 0.2$ results in proper convergence. In the case of a regular grid $l_{\mathrm{c}} = l_{\mathrm{grid}}$. By default in OPENRADIOSS the CFL condition uses $C_{\mathrm{CFL,fid}} = 0.9$, this is the same value as is used in LS-DYNA. It is investigated if a varied range of CFL conditions give identical results, the CFL constant values are evenly spaced in log space following:

$$C_{\mathrm{CFL}} = C_{\mathrm{CFL,fid}} 2^{n/4}, n \in \mathbb{Z}. \tag{2.81}$$

This is done for both beam and shell elements. For the shell elements $C_{\mathrm{CFL}} > 1.0703$ is problematic [14], similar for the beam elements $C_{\mathrm{CFL}} > 0.9$ is problematic. Values that are problematic result quickly in problems with the energy in the simulation. This is something that OPENRADIOSS quickly indicates, and the simulation is stopped. Because $C_{\mathrm{CFL}} = 0.9$ is just stable, it is not a good choice in general. This implies that

[14] This larger value than 1.0 is a bit surprising but is probably due to the way OPENRADIOSS rounds-off numbers to calculate the CFL condition.

a value of $C_{\mathrm{CFL}} \leq 0.75$ is recommended. It is noted that for each type of problem it is important to verify that this condition is good enough. Especially when shocks with high Mach numbers are present a smaller value will be required (e.g. shocks in air, FSI of air shock).

For the CFL condition only an investigation at the spatial resolution of 125 mm is performed, this corresponds to a total of 32 resolution elements in the length. When looking at the spatial distribution of the cantilever beam with different CFL constants there are no noticeable differences (not shown). Fig. 2.10 shows the determined natural frequencies for the cantilever beam. Across all natural frequencies the convergence of the natural frequencies is nearly perfect. Similarly, Fig. 2.11 shows that the vertical displacement and its scatter are well converged for the different CFL conditions. Overall, this indicates that a $C_{\mathrm{CFL}} \leq 0.75$ will produce good results in OpenRadioss.

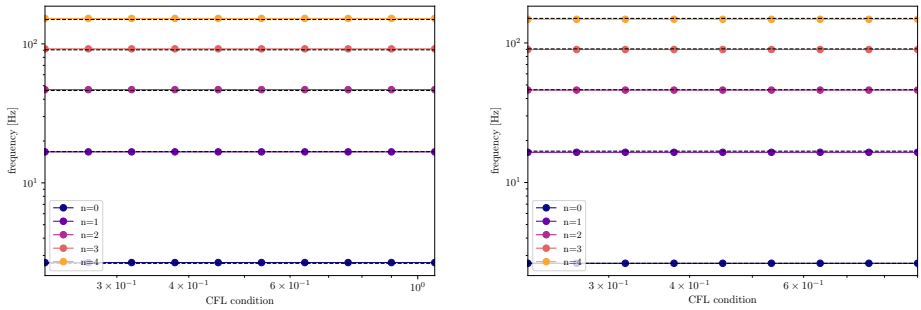


Figure 2.10 Comparison of the inferred natural frequencies for Zeng and Combescure (1998) shell (left) and Timoshenko (1921, 1922) beam elements (right) as a function of CFL condition and for different natural frequencies (different colours). Excellent convergence with the CFL condition is found, larger values than shown do not result in convergence of the result. Overall, it is recommended to use at most $C_{\mathrm{CFL}} = 0.9$. Given that this will be problematic for shocks, an overall recommendation is given of $C_{\mathrm{CFL}} \leq 0.75$ such that more non-linear and more rapid phenomena are also described correctly.

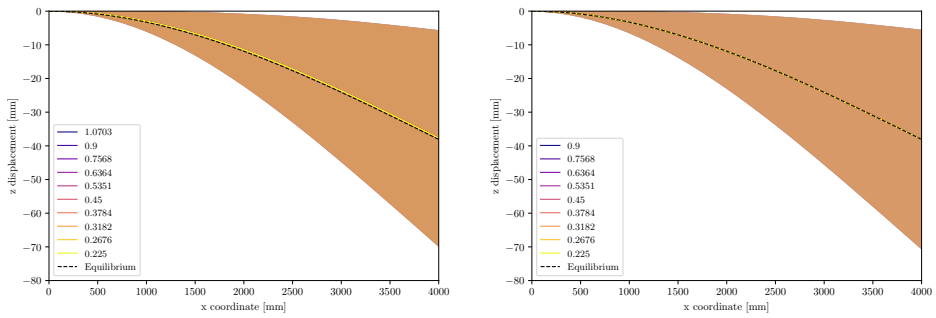


Figure 2.11 Comparison of the equilibrium displacement and the 16th and 84th percentile for the vertical displacement for Zeng and Combescure (1998) shell (left) and Timoshenko (1921, 1922) beam elements (right). The different colours indicate the different CFL condition. The results converge nearly perfect as a function of CFL condition. Larger values than shown do not result in convergence. Overall, this figure shows that at least $C_{\mathrm{CFL}} = 0.9$ should be used. It is recommended to at least use $C_{\mathrm{CFL}} \leq 0.75$ for good convergence in general.

An important note of the investigation of the (CFL) condition is that we only investigated a regular mesh with square shells. This means that the impact of the (CFL) condition is much bigger because all cells have the same time step. In a full ship or submarine model there are many elements with different sizes and only a very limited number of elements have the condition that they need to have a very small time step.

This means that for a (CFL) condition of $C_{\mathrm{CFL}} = 0.9$ we do not expect much problems for irregular models.

2.5.3 Comparison of different hourglass and shell elements formulations

In § 2.3.3 an overview of the different shell formulations in OPENRADIOSS was given. Here a direct comparison at six different resolutions is performed for all six shell element formulations in OPENRADIOSS. Fig. 2.12 shows a convergence test for the Belytschko and Tsay (1983) shell elements with the four different hourglass penalisations. This demonstrates that the different hourglass penalisations produce nearly identical results that are hard to distinguish. This is not surprising because $\sigma_{\mathrm{y}} \approx \infty$ which means that $I_{\mathrm{shell}} = 1$ is identical to $I_{\mathrm{shell}} = 3$ because the second term of equations (2.66) and (2.67) reduce to equations (2.62) and (2.63). Furthermore, the cantilever beam also only has a small displacement, this means that the penalisation model without orthogonality ($I_{\mathrm{shell}} = 2$) and the penalisation model with orthogonalisation for warped elements ($I_{\mathrm{shell}} = 4$) are not expected to show any difference with respect to $I_{\mathrm{shell}} = 1$. Still for all the four different penalisation methods the convergence of the shell elements is not perfect and as explained in § 2.3.3 they are unable to converge to the correct solution of the cantilever beam benchmark. Despite this the numerical solution remains within 7 per cent of the analytical solution. That all four hourglass penalisation methods produce almost identical results applies to most linear tests, however, for more complicated models this conclusion might not apply and it should be considered which of the four hourglass penalisation methods is most appropriate.

Fig. 2.13 compares the three different element formulations in OPENRADIOSS. From top to bottom Fig. 2.13 shows $I_{\mathrm{shell}} = 4$, $I_{\mathrm{shell}} = 12$, and $I_{\mathrm{shell}} = 24$. $I_{\mathrm{shell}} = 4$ corresponds to the most advanced hourglass penalisation with the Belytschko and Tsay (1983) shell elements. This plot shows that the model converges when the amount of resolution elements is increased from $n = 4$ to $n = 32$. At $n = 64$ the Belytschko and Tsay (1983) shell elements deviate again more from the analytical solution and at $n = 128$ they are more converged again. This shows that the convergence of the Belytschko and Tsay (1983) still deviates around 10 per cent from the correct solution. $I_{\mathrm{shell}} = 12$ corresponds to the fully-integrated Batoz and Dhatt (1990) shell elements with the physical hourglass formulations following Belytschko and Leviathan (1994) and Belytschko et al. (1984). $I_{\mathrm{shell}} = 12$ converges but shows too much stiffness which causes it to deviate around 13 per cent from the analytical solution. The result of being too stiff is well converged from 64 and 128 element. $I_{\mathrm{shell}} = 24$ corresponds to the Zeng and Combescure (1998) shell formulation using physical hourglass formulations. Similar to $I_{\mathrm{shell}} = 12$, the solution of this shell formulation converges. $I_{\mathrm{shell}} = 24$ shell elements are slightly too stiff which cause them to deviate consistently by 5 per cent from the analytical solution. Overall, the best shell element formulation to simulate the cantilever beam correspond to $I_{\mathrm{shell}} = 24$ or the Zeng and Combescure (1998) shell elements. These shell elements converge well and converge quite close to the analytical solution. However, a deviation of 5 per cent is still significant.

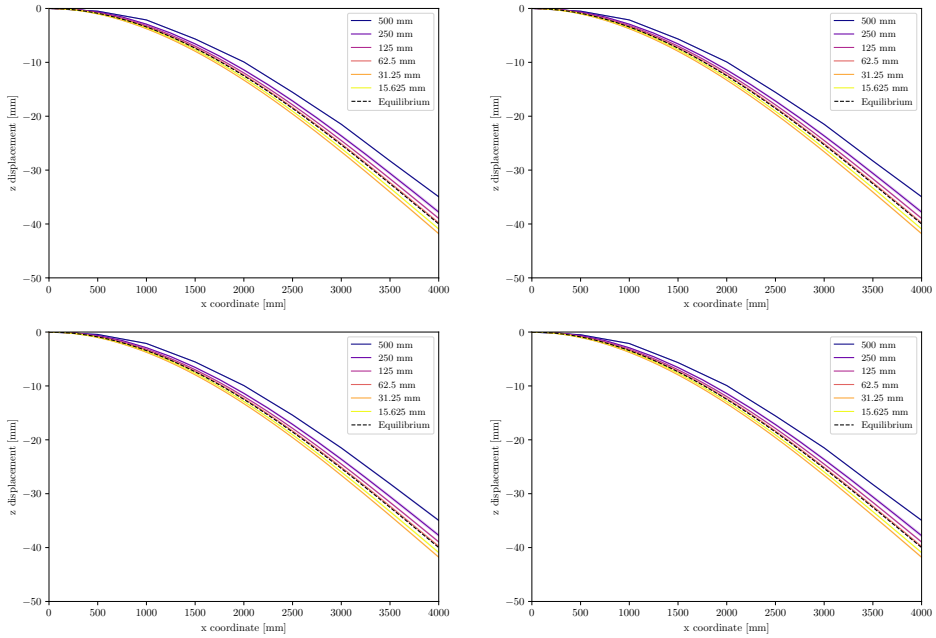


Figure 2.12 Comparison of the different hourglass penalisation methods using the Belytschko and Tsay (1983) shell elements for different numerical resolutions (colours). From top left to bottom right, $I_{\text{shell}} = 1$, $I_{\text{shell}} = 2$, $I_{\text{shell}} = 3$, and $I_{\text{shell}} = 4$. $I_{\text{shell}} = 1$ uses the Kosloff and Frazier (1978) and Flanagan and Belytschko (1981) visco-elastic hourglass formulation. $I_{\text{shell}} = 2$ uses the Kosloff and Frazier (1978) and Flanagan and Belytschko (1981) visco-elastic hourglass formulation without orthogonality following Hallquist (Altair, 2022a), $I_{\text{shell}} = 3$ uses a modified Flanagan and Belytschko (1981) elastoplastic hourglass formulation which set a minimum value to the hourglass correction based on the material yield. $I_{\text{shell}} = 4$ uses the Kosloff and Frazier (1978) and Flanagan and Belytschko (1981) visco-elastic hourglass formulation which includes a correction for warped elements such that the hourglass formulation remains orthogonal. For the cantilever beam benchmark, the four different hourglass penalisation methods give nearly identical results but the convergence is not amazing because even at high resolution the simulation can be off by around 7 per cent (orange line) and is better for slightly higher and lower resolutions (yellow and dark orange) that have a deviation of around 3 per cent and 1 per cent.

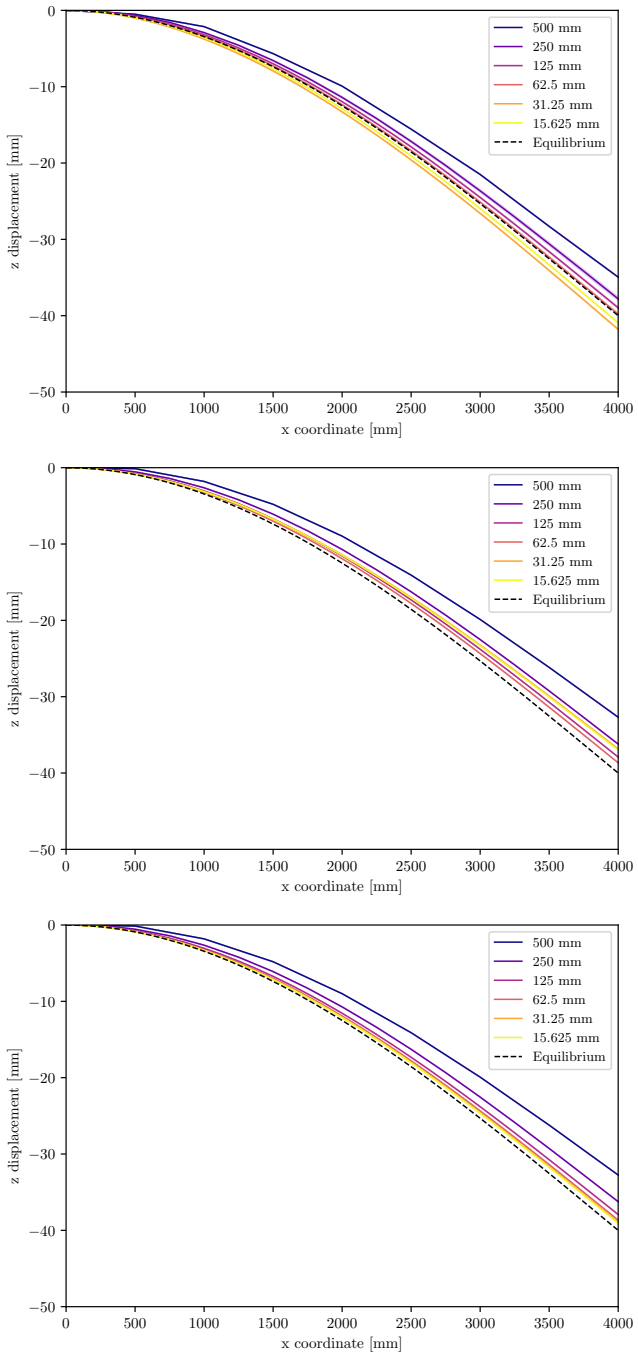


Figure 2.13 Comparison of the three different shell formulations in OpenRadioss for different numerical resolutions (colours). From top to bottom, $I_{\text{shell}} = 4$, $I_{\text{shell}} = 12$, and $I_{\text{shell}} = 24$. $I_{\text{shell}} = 4$ corresponds to the Belytschko and Tsay (1983) shell elements that include orthogonal viscoelastic hourglass corrections following the Kosloff and Frazier (1978) and Flanagan and Belytschko (1981) formulation which take into account warping of elements. $I_{\text{shell}} = 12$ corresponds to the fully-integrated Batoz and Dhatt (1990) shell elements with the physical hourglass formulations following Belytschko and Leviathan (1994) and Belytschko et al. (1984). $I_{\text{shell}} = 24$ corresponds to the Zeng and Combescure (1998) shell formulation using physical hourglass formulations. An important difference between $I_{\text{shell}} = 4$ and $I_{\text{shell}} = 12 = 24$ is that $I_{\text{shell}} = 12$ and 24 converge towards a single solution as the resolution is increased while $I_{\text{shell}} = 4$ is still not converged. Furthermore, the three different shell formulations have different stiffness, the $I_{\text{shell}} = 4$ elements are the least stiff while $I_{\text{shell}} = 12$ is most stiff and $I_{\text{shell}} = 24$ is between. For the three different shell element formulations, $I_{\text{shell}} = 24$ or Zeng and Combescure (1998) shell elements converges and agree best with the analytical solution.

2.5.4 Thickness integration

In § 2.3.4 the way thickness integration of shell elements is done in OPENRADIOSS is explained. Here the importance of the number of integration points in investigated. To investigate this a simultaneous variation of the resolution and the number of thickness integration points is performed. Fig. 2.14 shows the impact of the number of thickness integration points and the numerical resolution. As expected, based on § 2.3.4 the result quickly converges when the number of integration points is 2 or higher. Note that the result here is for a linear problem, which means that all quantities involved are either linear or quadratic. Gauss-Lobatto quadrature is accurate for any polynomial $2n-1$ which means that the result here does not generalise for non-linear problems.

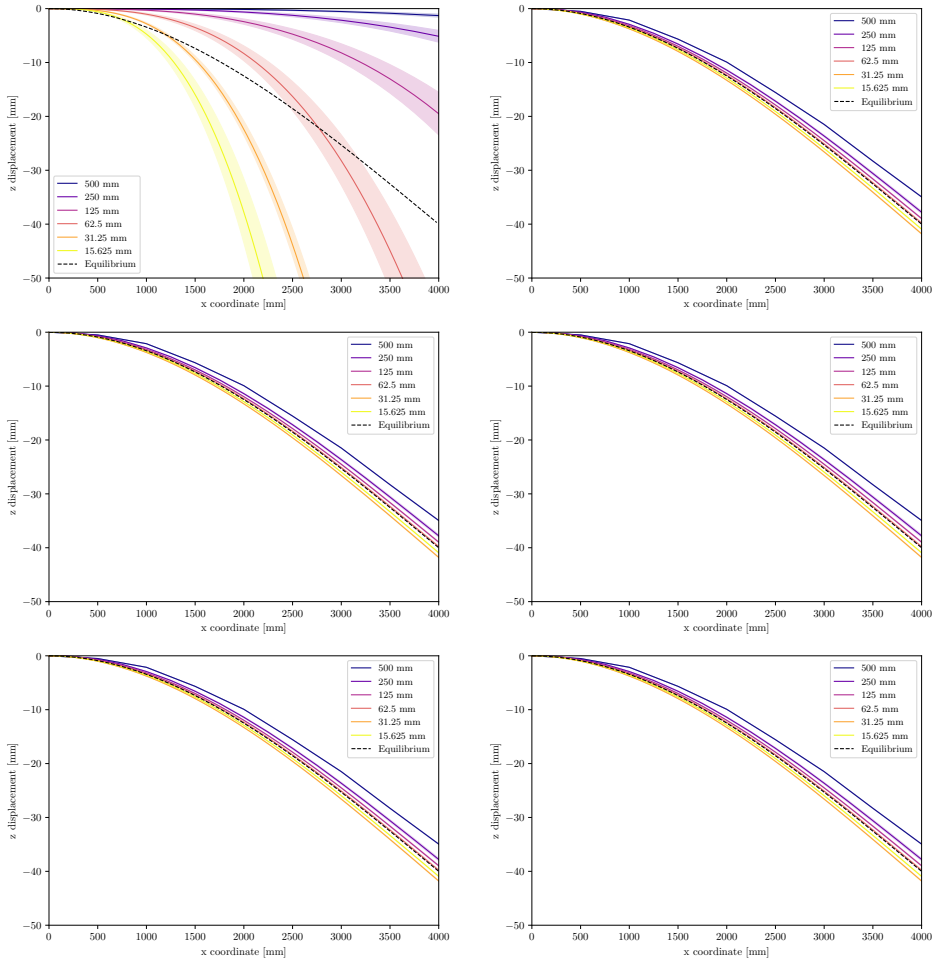


Figure 2.14 Comparison of the importance of number of thickness integration points (different columns/rows) and the numerical resolution (different colours) for the cantilever beam benchmark using Belytschko and Tsay (1983) shell elements. The Kosloff and Frazier (1978) and Flanagan and Belytschko (1981) visco-elastic hourglass formulation is used which includes a correction for warped elements such that the hourglass formulation remains orthogonal ($I_{\text{shell}} = 4$). The different panels show 1 (membrane behaviour), 2, 3, 5, 7 and 9 integration points using Gauss-Lobatto quadrature. The result is converged from $N = 2$ integration points as expected from theory.

2.5.5 Force on the side of the cantilever beam

In the examples above only normal forces on the horizontal surface were place. Here a rotated force is applied to test shell elements in their own plane. Fig. 2.15 shows the procedure used in the previous sections in red, here all the last nodes obtain a force. A different test is rotating the applied force by 90° and only forcing the force on a single node, this is demonstrated by the blue arrow. This case is called the alternative forced cantilever.

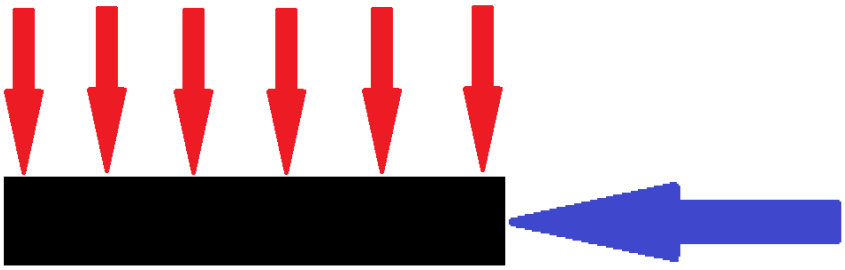


Figure 2.15 Schematic view of the cantilever beam and the applied force. The red arrows show how the force is applied in the previous sections and the blue arrow indicates how the force is applied in this subsection.

The cross section has a (much) larger inertia in the direction of the blue loading than in the direction of the red loading. This means that the expected frequency is orders of magnitudes larger for the blue case than in the case of the red case, it will be different by $b/D \approx 125$. We also increased the force by a factor of 10^4 such that the displacement remains of the order of 20 mm.

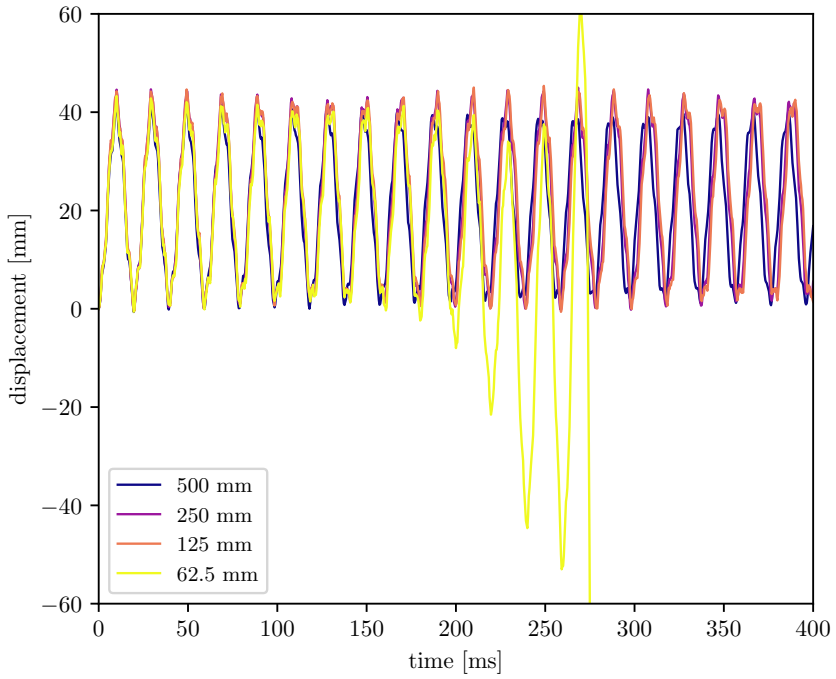


Figure 2.16 Comparison of the displacement at the end of the alternative forced cantilever beam for different numerical resolutions (colours) using Zeng and Combescure (1998) shell elements. For the lowest resolution simulations the solution remains stable while for the highest resolution simulation the simulation diverges at only 200 ms.

Fig. 2.16 shows the 400 ms of the simulation with the alternative forced cantilever beam using Zeng and Combescure (1998) shell elements. For the three lowest resolution simulations the solution remains relatively stable, while the highest resolution simulation quickly deviates at around 200 ms. Simulations for two- and four-times higher resolution were also performed and resulted in diverging results, these simulations are not shown here.

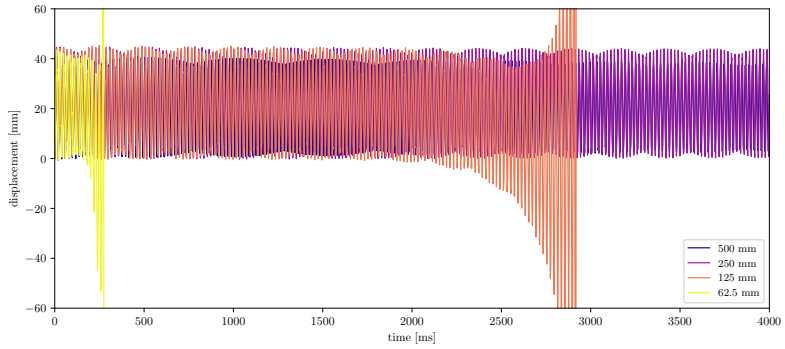


Figure 2.17 Same as Fig. 2.16 but for a longer time. The simulation with a resolution of 125 mm starts deviation at around 2000 ms. This indicates that the simulations of the alternative forced cantilever beam are unstable when integrated over long times.

The theoretical frequency is around 125×2.6 Hz ≈ 325 Hz. Fig. 2.16 shows that within 400 ms there are 20.5 periods, therefore the frequency is given by $2\pi \times 20.5/400$ ms $=$ 322 Hz. This shows that the frequency in the simulation is in agreement with the theoretical frequency.

Fig. 2.18 shows results when we perform the alternative forced cantilever beam for different CFL conditions. It is found that a smaller CFL condition can indeed improve the stability of the solution. But that a CFL condition of 0.45 still produces instability after some 4000 ms.

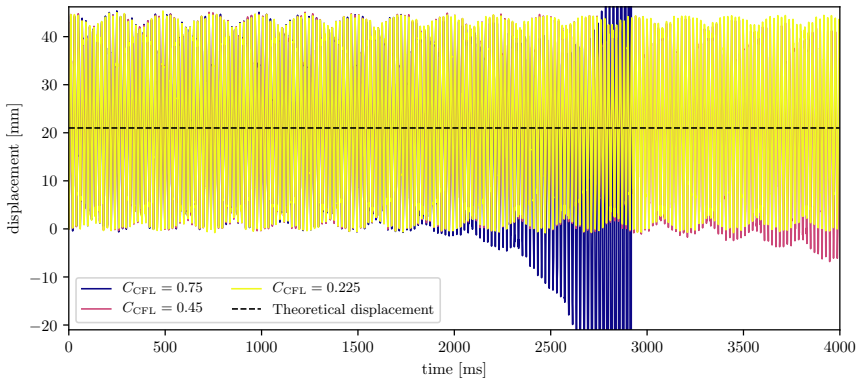


Figure 2.18 Comparison of the displacement at the end of the alternative forced cantilever beam for different CFL conditions (colours) using Zeng and Combescure (1998) shell elements at a resolution of 125 mm. Reducing the CFL conditions improves the convergence of the results.

Fig. 2.19 shows the same as Fig. 2.18 but for a larger element resolution. This again demonstrates that the stability of the solution can be improved when the CFL condition is decreased but that it is hard for the solution to remain stable for many cycles.

Fig. 2.20 shows the convergence at a spatial resolution of 125 mm and demonstrates that this stability issue is present also for the Belytschko and Tsay (1983) shell elements.

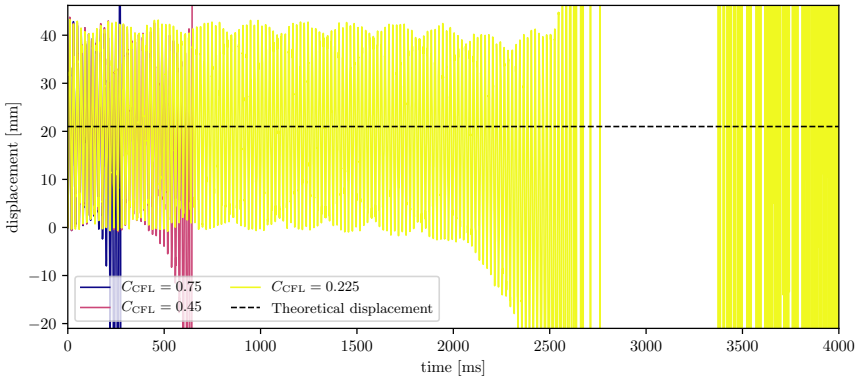


Figure 2.19 Comparison of the displacement at the end of the alternative forced cantilever beam for different CFL conditions (colours) using Zeng and Combescure (1998) shell elements at a resolution of 62.5 mm. Reducing the CFL conditions improves the convergence of the results.

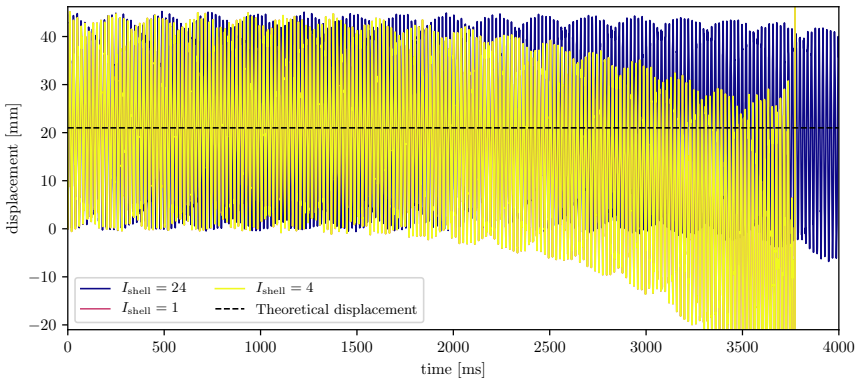


Figure 2.20 Comparison of the displacement at the end of the alternative forced cantilever beam for different shell elements at a resolution of 125 mm and a CFL condition of 0.45. 24 correspond to the Zeng and Combescure (1998) shell elements and 1 and 4 correspond to the Belytschko and Tsay (1983) shell element with different hourglass penalisation methods. At this resolution the Zeng and Combescure (1998) shell elements obtain a less divergent result than the Belytschko and Tsay (1983) shell elements.

The question is whether it is surprising that the oscillations deviate so much and the system obtains additional energy. OPENRADIOSS uses central difference for the time integration. Central differences has the problem that it does not conserve energy. As shown in Springel (2005), this can either mean that the system slowly loses energy (dissipation) or that it gains energy. The central difference time integration in OPENRADIOSS mainly breaks down when it gains energy rather than when it loses energy. Fig. 2.4 showed that at low resolutions dissipation of energy takes place. To solve the problem of losing and gaining energy it will be required for the time integration to conserve energy by construction. Symplectic time integration schemes always conserve the total energy of the system because they are designed to do so. These are time integration schemes like leapfrog integration (Birdsall and Langdon, 1985) or Verlet (1967) integration.

2.6 Damped cantilever beam

In this section we again focus on the cantilever beam in which the force is imposed perpendicular to the plate of the cantilever beam (Red case of Fig. 2.15).

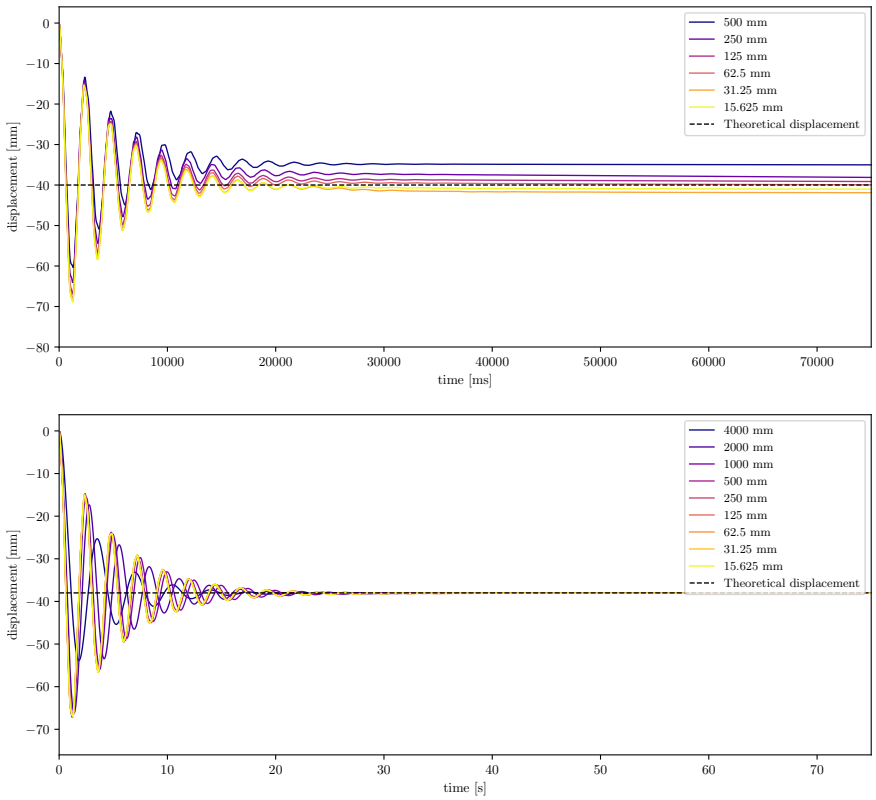


Figure 2.21 Comparison of the vertical displacement at the edge of the cantilever beam with damping for Belytschko and Tsay (1983) shell (top) and Timoshenko (1921, 1922) beam (bottom) elements. We find that the static displacement is not converged for the three lowest resolutions when using shell elements, contrary, when beam elements are used the static displacement is converged already when 1 element is used.

To investigate specifically the static solution of the cantilever beam we apply Rayleigh mass and stiffness damping to the problem. This damping results in a relatively quick decrease of the amplitude such that we can compare well with the static solution at different numerical resolutions. In Fig. 2.21 we show the time evolution of the displacement for shell and beam elements. We find that the beam elements are very well converged in terms of their static vertical displacement[15] and find that the displacement is converged when a single beam element is used. The shell elements are not as well converged at low element resolutions and require at least a resolution of 125 mm in order for the static vertical displacement to be converged.

In Fig. 2.22 we show the static displacement for all element resolutions in the simulations of the cantilever beam. We find again excellent convergence for the beam elements which are very well converged even when damping is included. The convergence for shell elements is slightly worse and they require a slightly higher resolution of around 125 mm to converge. Overall, this result shows that there is correct convergence when damping is used.

Fig. 2.23 shows the reduced Chi-squared value of the different resolutions as a more quantitative way of quantifying how good the convergence is between the theoretical solution and the simulation itself. We find that the beam elements show excellent agreement with the theoretical values at their node positions and the accuracy is quite

[15] Because of a slightly different variable set, the mean displacement for the beam elements is slightly smaller than 40.

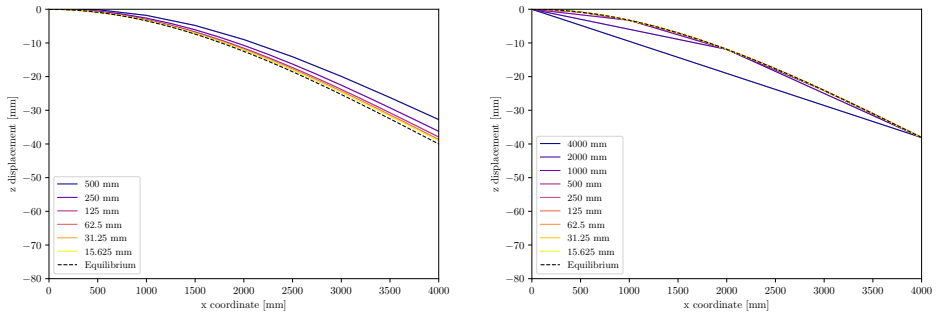


Figure 2.22 Comparison of the static vertical displacement of the analytical solution and the simulations at different resolutions (different colours) for Belytschko and Tsay (1983) shell (left) and Timoshenko (1921, 1922) beam (right) elements. We find that beam elements converge very well even at the lowest resolutions. Shell elements require slightly higher resolutions of 125 mm to have good convergence. This results shows that the results keep converging when there is damping.

close to machine accuracy of float numbers of around 10^{-8}. The overall convergence of the exact shape converged very rapidly with almost $\propto l_{\text{res}}^5$. The shell elements show less strong convergence and show only converged up to a few times 10^{-3}.

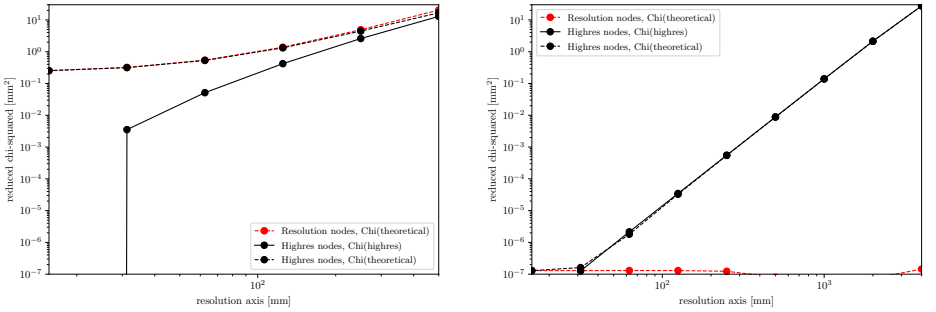


Figure 2.23 Comparison of the reduced chi-squared as a function of numerical resolution for Belytschko and Tsay (1983) shell (left) and Timoshenko (1921, 1922) beam (right) elements with damping. The convergence for beam elements is very good, for shell elements the convergence is less good but is close to 10^{-3}.

2.7 Cantilever beam with different damping

In Fig. 2.24 we show simulations with slightly different mass damping for both shell and beam elements at a spatial resolution of 125 mm. We see that both element types converge to the desired mean displacement. This indicates that the result is independent of the mass damping. All simulations were performed with a small stiffness damping.

To not have oscillations at all times, instead of continuing the problem like it is, I take a slightly different approach. For all nodes we will include Rayleigh damping. Specifically, we will only consider Rayleigh mass damping as the stiffness damping is not desired in the case of the cantilever beam because we want to find the equilibrium solution. In general, it is possible to find the Rayleigh mass damping that is around the critical value ($\xi = 1$) by calculating the Rayleigh mass damping as

$$\alpha = \frac{2\xi}{2\omega_n}, \tag{2.82}$$

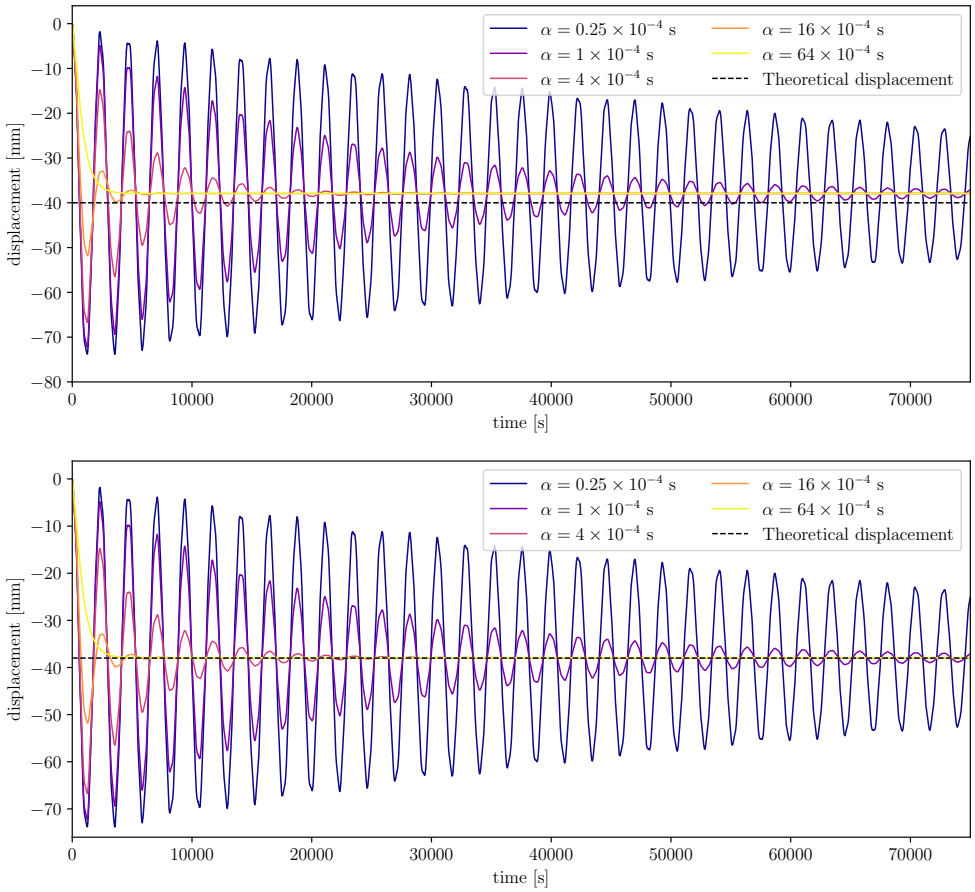


Figure 2.24 Comparison of the vertical displacement at the edge of the cantilever beam with damping for Belytschko and Tsay (1983) shell (top) and Timoshenko (1921, 1922) beam (bottom) elements with damping. We find that the static displacement is converged for simulations with different mass damping.

where ω_n is the natural frequency, this means that we would expect the Rayleigh mass damping to be around $\alpha = 2/(2\pi \cdot 2.6 \text{ Hz}) = 0.12$ s. By performing tests, we found this value to be almost a factor of 1000 smaller and therefore we use these values around $\alpha = 4 \times 10^{-4}$ s instead. For the simulation of a spatial resolution of 125 mm we perform a few simulations with different Rayleigh mass damping factors around this fiducial value. We space them with factors of 4. In general, the damping shows the correct trend.

2.8 Conclusions

- A cantilever beam benchmark was created and used to verify the different shell and beam element formulations.
- Timoshenko (1921, 1922) beam and Zeng and Combescure (1998) shell elements reproduce the correct natural frequencies, equilibrium position, and normalisation for the cantilever beam benchmark. It is found that the natural frequency up to $n = 5$ can be predicted well.
- A CFL condition that is slightly stricter than the fiducial value of 0.9, but instead is 0.75 improves the stability for a regular mesh and allows the cantilever beam benchmark to be run correctly without divergence.
- The Zeng and Combescure (1998) shell formulation is found to produce the

most desired solution for the cantilever beam benchmark because; (i) it shows converging results, (ii) it converges closely to the desired analytical solution. The Batoz and Dhatt (1990) Q4γ24 shell element do not converge to the correct analytical solution, but they show clear convergence with a standard deviation from the solution. Lastly, the Belytschko and Tsay (1983) shell element shows that the result does not strictly converge in which a higher resolution means; a solution closer to the analytical solution. The Belytschko and Tsay (1983) shell element shows rather that the solution oscillates around the true solution and the amount of convergence is difficult to know in advance.

- An alternative cantilever beam was used that shows that the different shell element formulations have trouble finding a converging solution when the oscillations of the cantilever beam have a very small period.
- The cantilever beam benchmark was also performed with a damping factor which let the solution converge to the equilibrium state as expected.

3 Springs

This chapter verifies the simple spring elements in OPENRADIOSS. An extensive investigation of ideal springs and non-ideal springs is performed. These are the first steps to investigate spring elements that are used for UNDEX analysis.

3.1 Ideal spring

This section focuses on an ideal spring. The simulations performed are based on a spring system which has a mass, this means there are 2 nodes for the spring, one node for the orientation, and a single spring element. For a spring with a mass, it is required to consider the effective mass of the spring-mass system. A spring with mass m does not have a kinetic energy of $\frac{1}{2}mv^2$, where v is the velocity at the end of the spring. Only the very end of the spring will move with that velocity. The differential kinetic energy is given by:

$$\mathrm{d}K = \frac{1}{2}u^2\mathrm{d}m. \tag{3.1}$$

To get the total kinetic energy equation (3.1) needs to be integrated over the total length of the spring,

$$K = \int_{\text{spring}} \mathrm{d}K = \int_{\text{spring}} \frac{1}{2}u^2\mathrm{d}m. \tag{3.2}$$

An ideal spring is stretched homogeneously, which means that its mass distribution is uniform. Therefore $\mathrm{d}m = \frac{m}{l}\mathrm{d}s$, l is the length of the spring measured at time t and $\mathrm{d}s$ is the differential of distance. Assuming a homogeneous stretch also gives that the velocity is given by $u(s) = \frac{s}{l}v$. This means that the kinetic energy becomes:

$$K = \int_0^l \frac{1}{2}\left(\frac{s}{l}v\right)^2\left(\frac{m}{l}\right)\mathrm{d}s, \tag{3.3}$$

$$= \frac{m}{2l^3}v^2\int_0^l s^2\mathrm{d}s, \tag{3.4}$$

$$= \frac{1}{2}\frac{m}{3}v^2. \tag{3.5}$$

This means that the effective mass of the spring is simply $\frac{m}{3}$.

How to test single springs? There is just one way of testing a spring properly, this is by exerting a force along its direction and following the dynamic evolution of the spring. The EoM of a spring can simply be calculated by setting Newton's (1687) second law equal to Hooke's (1678) law and the EoM reduces to:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -\frac{k}{m_{\text{eff}}}x. \tag{3.6}$$

where k is the spring constant and m_{eff} is its effective mass. Based on this the frequency of the oscillation is given by $\omega = \sqrt{k/m_{\text{eff}}}$. And the homogeneous solution is given by:

$$x(t) = c_1\cos(\omega t + \phi). \tag{3.7}$$

If a force is imposed the EoM is given by:

$$m_{\text{eff}} \frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = -kx + F_0 \Theta(t). \tag{3.8}$$

where F_0 is the imposed force and $\Theta(t)$ is the Heaviside step function[1] (Abramowitz and Stegun, 1965). This can be reduced to:

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} + \frac{k}{m_{\text{eff}}} x = \frac{F_0}{m_{\text{eff}}} \Theta(t). \tag{3.9}$$

The particular solution of this equation is given by:

$$x(t) = c_2 \Theta(t), \tag{3.10}$$

$$= \frac{F_0}{k} \Theta(t). \tag{3.11}$$

Combining this with the homogeneous solution gives the general solution given by:

$$x(t) = c_1 \cos(\omega t + \phi) + \frac{F_0}{k} \Theta(t) \tag{3.12}$$

The boundary conditions of the problem are given by:

$$x(t = 0) = 0, \tag{3.13}$$

$$\frac{\mathrm{d}x}{\mathrm{d}t} (t = 0) = 0. \tag{3.14}$$

Imposing these conditions gives:

$$c_1 \cos(\phi) + \frac{F_0}{k} = 0, \tag{3.15}$$

$$-c_1 \omega \sin(\phi) = 0. \tag{3.16}$$

This directly implies that $\phi = 0$, therefore $c_1 = -\frac{F_0}{k}$. This means that the solution is given by:

$$x(t) = \frac{F_0}{k} \left(\Theta(t) - \cos(\omega t) \right). \tag{3.17}$$

Using this relation there are two ways that the ideal spring can be tested:

- Comparing the harmonic oscillations of the spring with the analytical expectations.
- Comparing the displacement with the imposed force.

In theory it is possible to create a single numerical experiment to test both, but this has the limitation that the displacement and force graph does not extend to large displacements. Because of this, two different numerical experiments are performed:

- A constant force is imposed on the edge of the spring, the spring will start oscillating with frequency ω and will have a displacement following equation (3.17). This set-up is ideal to check that the spring has the right frequency, and the displacement is as expected.
- A constant axial displacement is performed and while displacing the spring the total force is calculated and logged. This is ideal to reproduce the force-displacement graph over a large dynamical range and allows comparing the input spring constant with the observed spring constant.

[1] The Heaviside step function is a function that is 0 for $t < 0$ and 1 for $t \geq 0$.

3.1.1 Time evolution

Fig. 3.1 shows the time evolution of an ideal spring and compares it with the analytical predictions of equation (3.17). The agreement is excellent, and for the time range shown we cannot see a clear deviation in phase of the oscillation. There might be a minuscule difference in the amplitude that is difficult to quantify. A better comparison might be to check how well the numerical simulation performs in Fourier space. Based on equation (3.17) the Fourier transform has two components, one component that is constant and transforms to a Dirac delta function at $\nu = 0$, that is $\propto \delta(\nu)$. And a Dirac delta function at the frequency (and negative frequency) of the cosine, that is $\propto \delta(\nu - \omega)$.

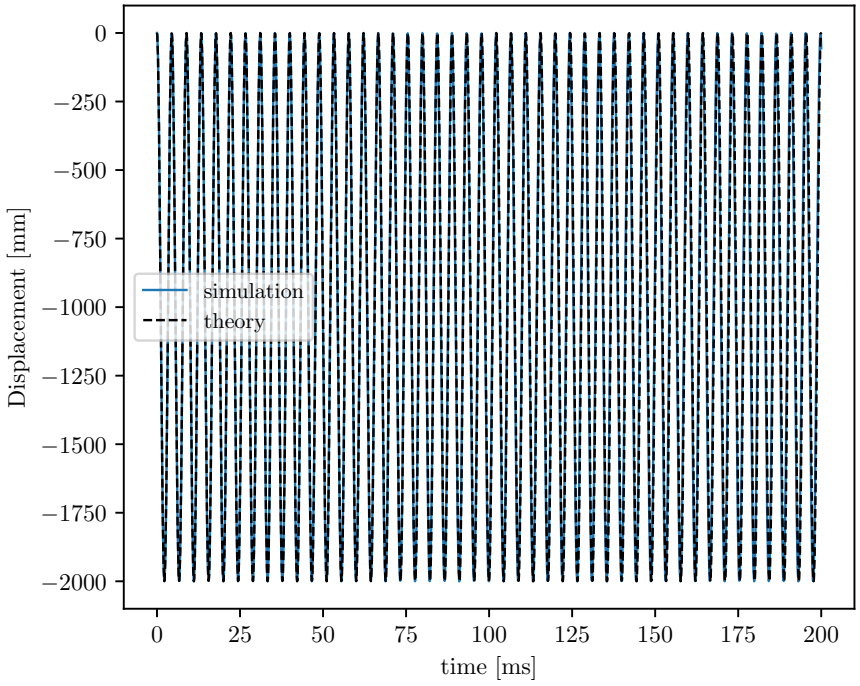


Figure 3.1 Comparing the time evolution of a simulated spring (blue) with the analytical solution (black dashed). The theoretical solution and the simulation are in excellent agreement.

Fig. 3.2 shows the FFT of the ideal spring system with an imposed force. Both the theoretical prediction and the numerical simulation impose a clear peak that match extremely well between both, this shows that OpenRadioss is working well in reproducing the eigenfrequency of the spring.

3.1.2 Displacement force relation

Fig. 3.3 compares the force-displacement relation of an ideal spring with the theoretical predictions. This demonstrates that the agreement is excellent between the theoretical model and the simulation, the difference is almost impossible to see in this plot and the lines are on top of each other.

The conclusion for ideal spring elements is that they produce the correct behaviour for both the displacement and the period.

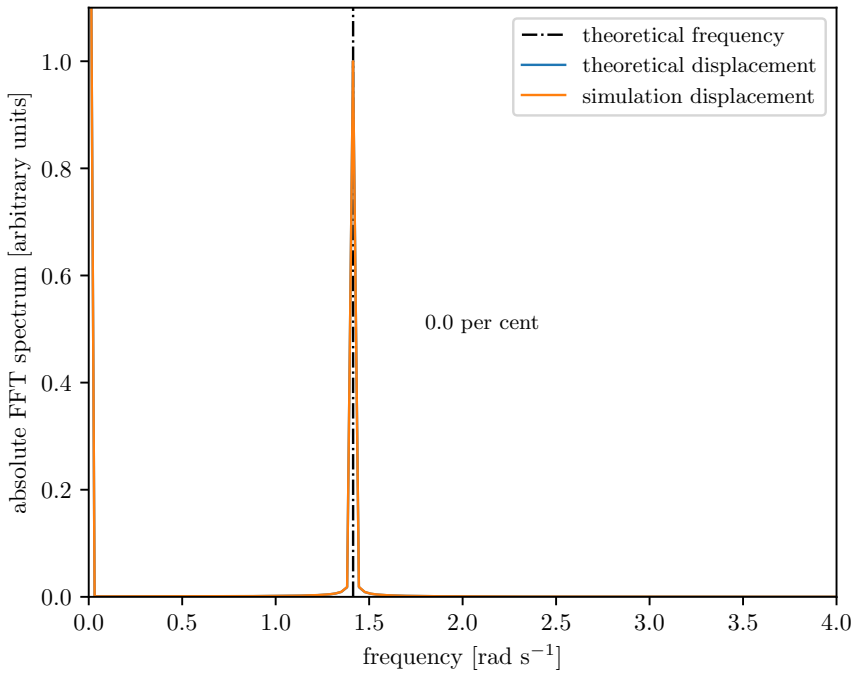


Figure 3.2 Compares the FFT spectrum of the theoretical prediction (blue and black dashed) and the numerical result of the simulation (orange). The theoretical frequency (black dashed) shows the input frequency ω and the theoretical displacement shows the Fourier transform of equation (3.17). The percentage shown in the figure shows the difference between the simulated and theoretical displacement, which is 0.0 per cent. Like the conclusion of Fig. 3.1, the frequency is in excellent agreement between the simulation and theoretical expectation.

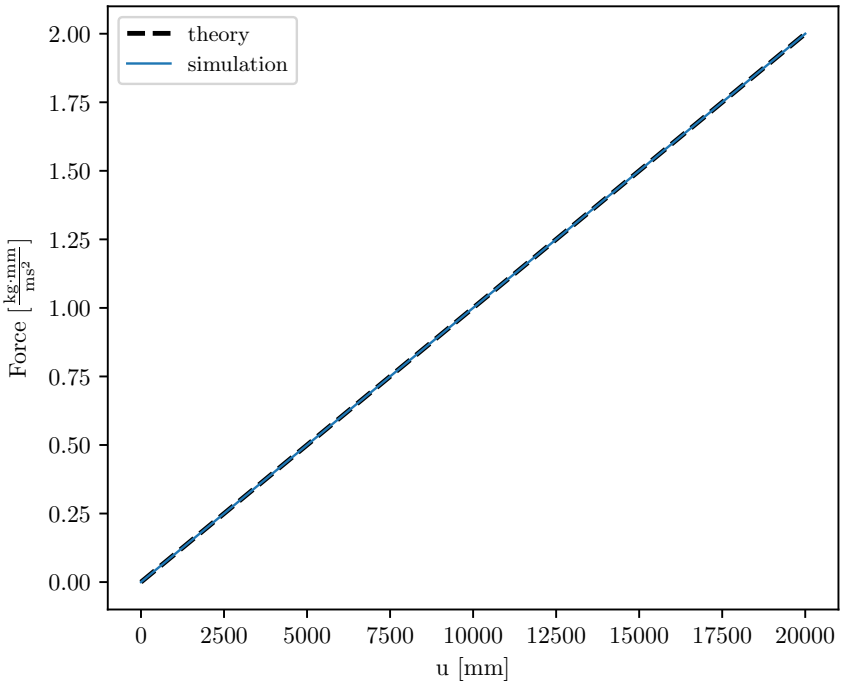


Figure 3.3 Comparing the theoretical (black dashed) force-displacement relation and the force-displacement relation from the numerical simulation (blue). The agreement is perfect between the theory and the simulation, the lines are on top of each other.

3.2 Damped ideal spring

Most spring elements also have a dash-pot damper that can be attached parallel to the spring element. When a damper is attached parallel to the spring its EoM changes to:

$$m_{\text{eff}}\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + c\frac{\mathrm{d}x}{\mathrm{d}t} + kx = F_0\Theta(t). \tag{3.18}$$

where c is the damping coefficient of the dash-pot damper. This means that the homogeneous solution reduces to:

$$x(t) = c_1 e^{-\omega t\xi}\sin(\omega t\sqrt{1-\xi^2} + \phi). \tag{3.19}$$

Where $\xi = \frac{c}{2m_{\text{eff}}\omega}$. When the damping coefficient is small, the term inside the sine reduces to:

$$\sqrt{1-\xi^2} \approx 1 - \frac{1}{2}\xi^2 - \frac{1}{8}\xi^4. \tag{3.20}$$

This means that when $\xi \ll 1$, equation (3.21) reduces to:

$$x(t) = c_1 e^{-\omega t\xi}\sin(\omega t + \phi). \tag{3.21}$$

This means that the change of frequency is small. For the damped spring, the only relevant measurement is seeing how quickly it damps. Because of this, the same set-up is used as for the ideal spring in § 3.1. Like in § 3.1 the general solution is:

$$x(t) = \frac{F_0}{k}\Theta(t) + c_1 e^{-\omega t\xi}\sin(\omega t\sqrt{1-\xi^2} + \phi). \tag{3.22}$$

Adding the two boundary conditions:

$$x(t=0) = 0, \tag{3.23}$$

$$\frac{\mathrm{d}x}{\mathrm{d}t}(t=0) = 0. \tag{3.24}$$

This results in

$$x(t) = \frac{F_0}{k}\left(\Theta(t) - e^{-\omega t\xi}\cos(\omega t\sqrt{1-\xi^2})\right). \tag{3.25}$$

For the damped spring the displacement will be damped, and this is the main interest of investigating the damped ideal spring. Fig. 3.4 shows the displacement of the theory and the simulation. This shows that the agreement is excellent between the theory and simulation. The damping changes the frequency, this still results in excellent agreement between the simulation and theory.

Fig. 3.5 shows the Fourier transform of the displacement, this clearly shows that both agree very well, there peak frequency agrees perfectly between the theory and simulation. The shapes of the Fourier transforms are almost on top of each other but show a small deviation between them that is not significant for the simulation outcome.

As a side note, in OpenRadioss the time step size is given by (Altair, 2022b)

$$\Delta t = \frac{\left(\sqrt{mk + c^2}\right) - c}{k}. \tag{3.26}$$

This implies that only a few time steps are required for a period of a spring, this cannot produce accurate results and because of that Altair (2022b) recommend to use a time step that is a factor of at least 5 lower. The simulations performed here already use a CFL condition of $1/10$. Therefore, the expectation would be that the correct solution is produced. As a side check, simulations with a CFL condition of $1/100$ (not shown) were performed and results in identical results as for the CFL condition of $1/10$.

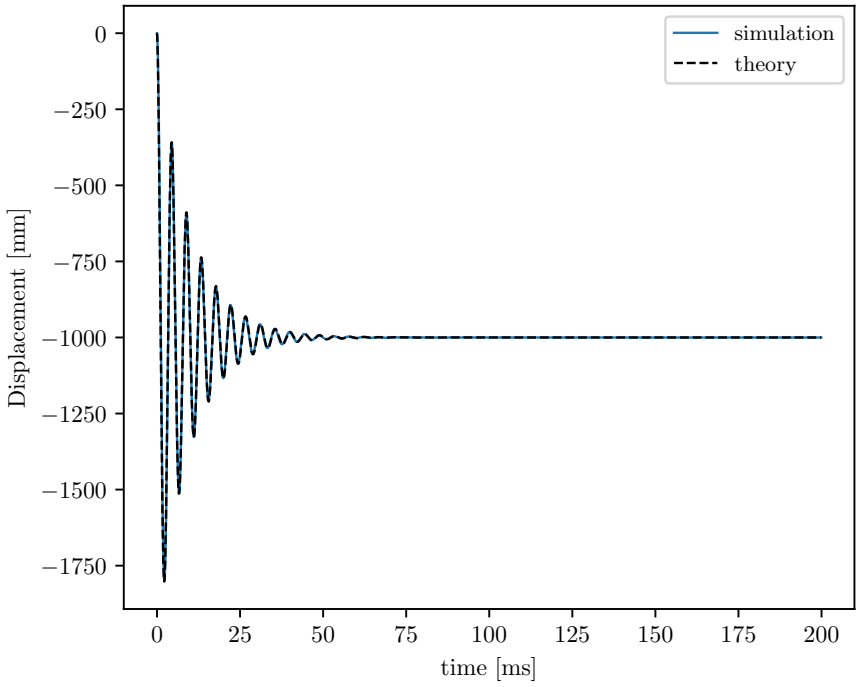


Figure 3.4 Comparing the theoretical displacement (black dashed) of a damped spring with the simulation of a damped spring (blue). The period and the decay agree perfectly.

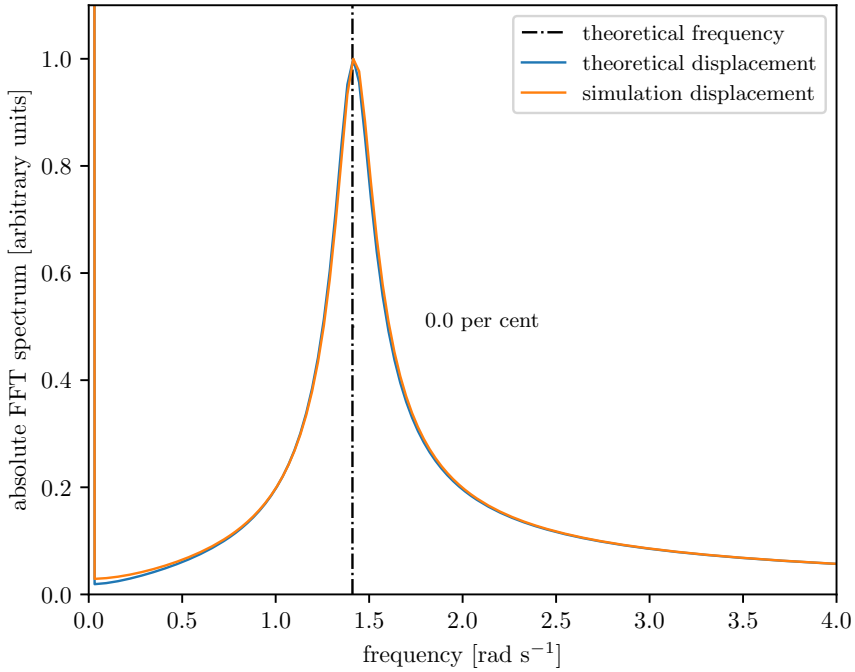


Figure 3.5 Comparison of the FFT signal of the displacement for the simulation (orange) and the theoretical prediction (blue and black dashed). The theoretical displacement (blue) shows the Fourier transform of equation (3.25). The theoretical frequency (black dashed) shows the theoretical frequency based on equation (3.25) which corresponds to $\omega\sqrt{1-\xi^2}$. The percentage shown in the figure shows the difference between the theoretical and simulated displacement. Like Fig. 3.2 the peak of the frequency agrees perfectly between the theoretical and simulated displacement.

3.3 Non-ideal spring

For non-linear springs with large contributions from non-linear terms it is important to only include terms that are producing periodic behaviour when periodic behaviour is investigated. This means that springs should have energies that have even parabolas. This means that non-linear terms can be

$$U_{\text{spring}} = \frac{1}{2}ku^2 + \frac{1}{4}k_2u^4 \tag{3.27}$$

or more general,

$$U_{\text{spring}} = \frac{1}{2}ku^2 + \sum_{i=1}^{\infty}\frac{1}{2i}k_iu^{2i}. \tag{3.28}$$

where k and k_i are spring constants that are either positive or zero. This implies that the force is given by:

$$F_{\text{spring}} = ku + \sum_{i=0}^{\infty}k_iu^{2i-1}. \tag{3.29}$$

Because equation (3.29) only has odd polynomials the force is always pointed to zero displacement.

3.3.1 Time evolution

When the time evolution is calculated the most important property is the frequency of the displacement. For a non-linear spring with only a single element, the energy is given by

$$U_{\text{spring}} = \int_0^x F_{\text{spring}}\mathrm{d}x, \tag{3.30}$$

where $f(x)$ is the nonlinear function. The assumption is made that the non-linear spring has only a single term and the term can be expressed as

$$F_{\text{spring}} = kx^{2n-1}, \tag{3.31}$$

where $n \geq 1$ and k is the spring constant with unit $\mathrm{N\ m^{1-2n}}$. The spring energy is then given by:

$$U_{\text{spring}} = \frac{k}{2n}x^{2n}. \tag{3.32}$$

This means that the energy of the system at maximal displacement is given by:

$$E_{\text{system}} = \frac{k}{2n}a^{2n}. \tag{3.33}$$

where a is the maximal displacement during a periodic oscillation, this corresponds to the amplitude of the oscillation. The kinetic energy of the system is given by $\frac{1}{2}mv^2$. This means that

$$E_{\text{system}} = U_{\text{spring}} + E_{\text{kinetic}}, \tag{3.34}$$

can be written as

$$\frac{1}{2}mv^2 = \frac{k}{2n}\left(a^{2n} - x^{2n}\right). \tag{3.35}$$

Reordering gives

$$v = \sqrt{\frac{k}{2mn}\,(a^{2n} - x^{2n})}. \tag{3.36}$$

This can simply be written as

$$\frac{1}{v} = \frac{\mathrm{d}t}{\mathrm{d}x} = \sqrt{\frac{nm}{k(a^{2n} - x^{2n})}}. \tag{3.37}$$

This equation can be integrated over a quarter of its period as

$$\frac{p}{4} = \int_0^a \frac{\mathrm{d}t}{\mathrm{d}x}\mathrm{d}x = \sqrt{\frac{nm}{k}} \int_0^a \frac{\mathrm{d}x}{\sqrt{a^{2n} - x^{2n}}} = \frac{1}{a^{n-1}}\sqrt{\frac{nm}{k}} \int_0^1 \frac{\mathrm{d}x}{\sqrt{1 - x^{2n}}}. \tag{3.38}$$

The integral has a standard solution given by:

$$\frac{C_n}{4\sqrt{n}} = \int_0^1 \frac{\mathrm{d}x}{\sqrt{1 - x^{2n}}} = \frac{\sqrt{\pi}\Gamma\left(1 + \frac{1}{2n}\right)}{\Gamma\left(\frac{n+1}{2n}\right)}, \tag{3.39}$$

where $\Gamma(x)$ is the Gamma function. This means that the solution for the period is given by

$$p = \frac{1}{a^{n-1}}\sqrt{\frac{m}{k}}\frac{4\sqrt{n\pi}\Gamma\left(1 + \frac{1}{2n}\right)}{\Gamma\left(\frac{n+1}{2n}\right)} = \frac{C_n}{a^{n-1}}\sqrt{\frac{m}{k}} \tag{3.40}$$

C_n can be calculated and gives the following result for the first few n, $C_1 = 2\pi$, $C_2 = 7.416299$, $C_3 = 8.413093$, $C_4 = 9.308741$ and for large n the expression reduces to $C_n = 4\sqrt{n}$. In this case the frequency can be determined to be

$$f = \frac{1}{p} = \frac{a^{n-1}}{C_n}\sqrt{\frac{k}{m}}. \tag{3.41}$$

The way to interpretate this equation is as follows, m is the mass of the spring, $a^{n-1}\sqrt{k}$ is the classical spring constant that has the units of $\mathrm{kg}^{1/2}\mathrm{s}^{-1}$. Combined $a^{n-1}\sqrt{k/m}$ gives a frequency but due to the non-linearity an additional correction factor is required. C_n is the correction factor for the normal frequency. This correction factor depends only on the order of the non-linear spring in equation (3.31).

Fig. 3.6 shows the time evolution of the non-ideal spring with a cubic function versus time. It is clear from this figure that the periodic behaviour of the non-linear spring produces regular cycles as would be expected. There is no analytical solution shown because the analytical solution is almost impossible to determine.

Fig. 3.7 compares the FFT of the non-linear spring with the analytical frequency of equation (3.41). Like ideal springs the frequency of the simulation is smaller by around 8 per cent.

Next a septic polynomial is considered for the periodic oscillation. Here it is assumed that $F_{\mathrm{spring}} = kx^7$. Fig. 3.8 shows the periodic behaviour of this oscillation.

Fig. 3.9 compares the FFT of the non-linear septic spring with analytical frequency of equation (3.41). Here we see a more drastic difference in frequency of around 700 per cent. Reflecting on this problem, the centre of the potential of the spring is not actually in the centre of this problem. Because we displace by a constant force the potential is passing through only one side of the potential and not the other. This means that the derived frequency is not applicable to our problem. We must come up with a different problem to test exactly this frequency.

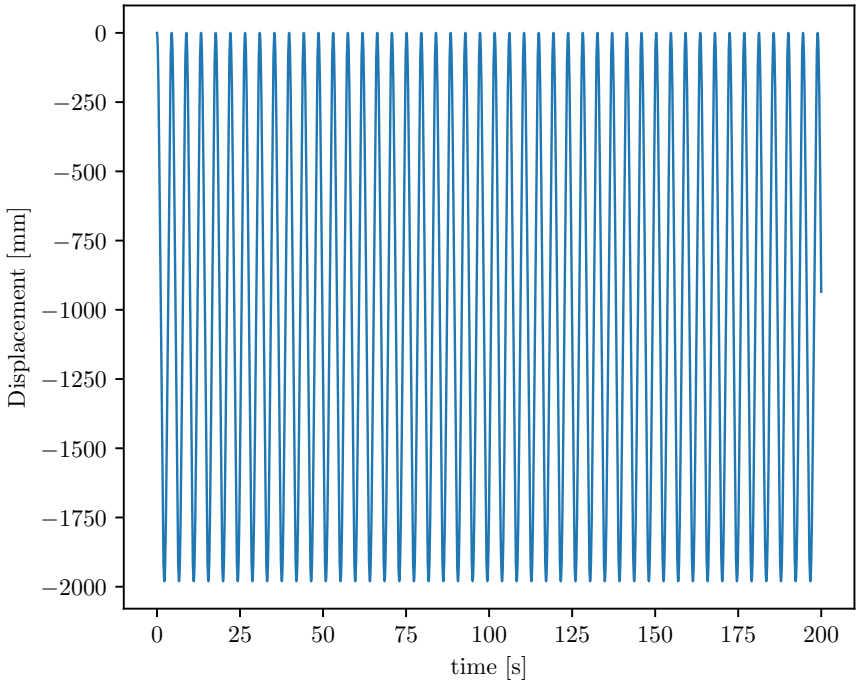


Figure 3.6 Non-linear springs show periodic behaviour when a force is imposed. As expected, the spring produces a periodic oscillation. No comparison is done with a theoretical solution because it is not possible to solve the problem analytically.

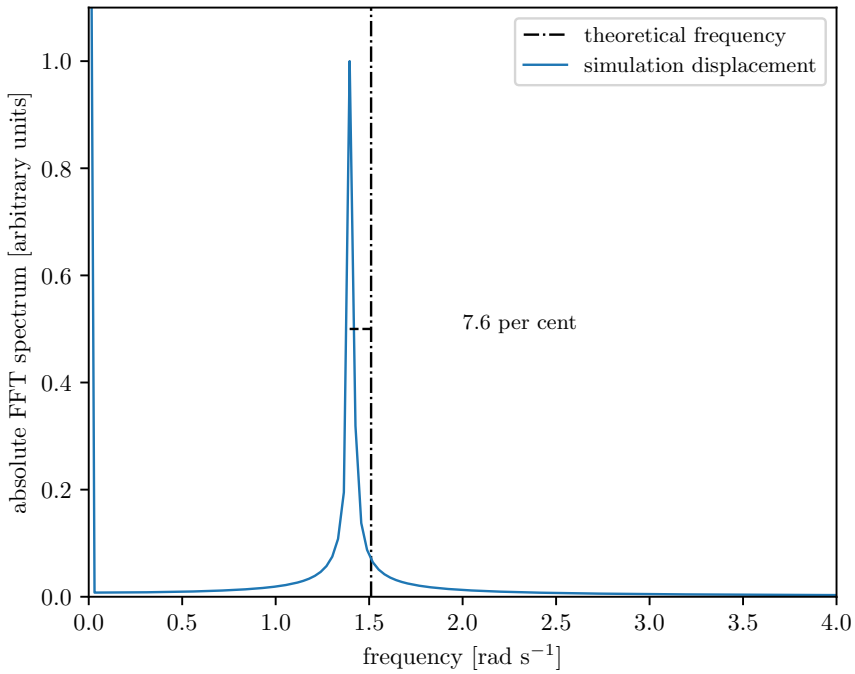


Figure 3.7 Comparison of the theoretical frequency of the spring following equation (3.41) with the FFT of the displacement of the simulation. This shows the same simulation as in Fig. 3.6. The non-linear springs following a $F = kx^3$ relation has a frequency of around 8 per cent smaller than the theoretical expected value.

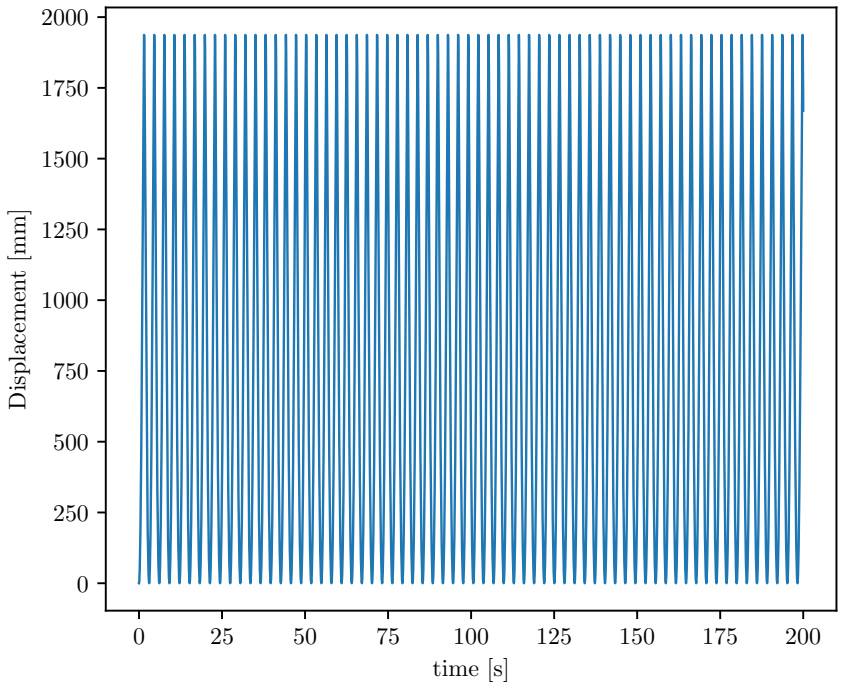


Figure 3.8 Non-linear springs following a septic polynomial shows periodic behaviour when a force is imposed. The spring produces a periodic oscillation. No comparison is done with a theoretical solution because it is not possible to solve the equation analytically.

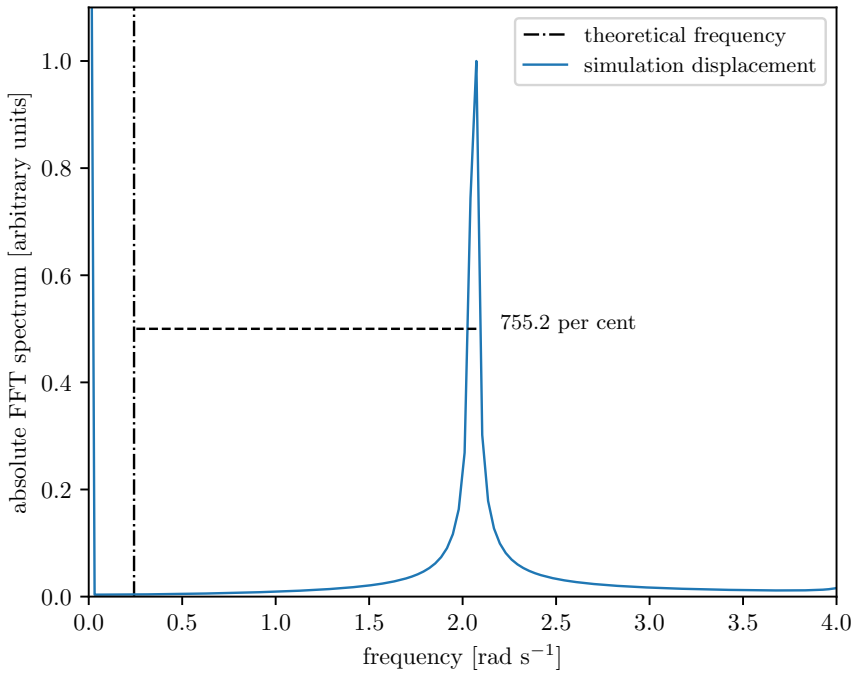


Figure 3.9 Comparison of the theoretical frequency of the spring following equation (3.41) with the FFT of the displacement of the simulation. The non-linear springs following a $F = kx^7$ relation has a frequency of around 750 per cent larger than the theoretical expected value.

3.3.2 Constant extension

For non-ideal springs that are just extended, the additional requirement of producing a periodic behaviour is not required and springs can be extended to any desired shape. Fig. 3.10 shows a smooth function of the form

$$F = A_1(ax^2 + bx). \tag{3.42}$$

where a, b are constants with units mm^{-2} and mm^{-1}. And A_1 is the unit that sets the force scale. Fig. 3.10 shows excellent agreement between the input function and the output function. Note that in this subsection the unit of force is given by kg mm ms^{-2}.

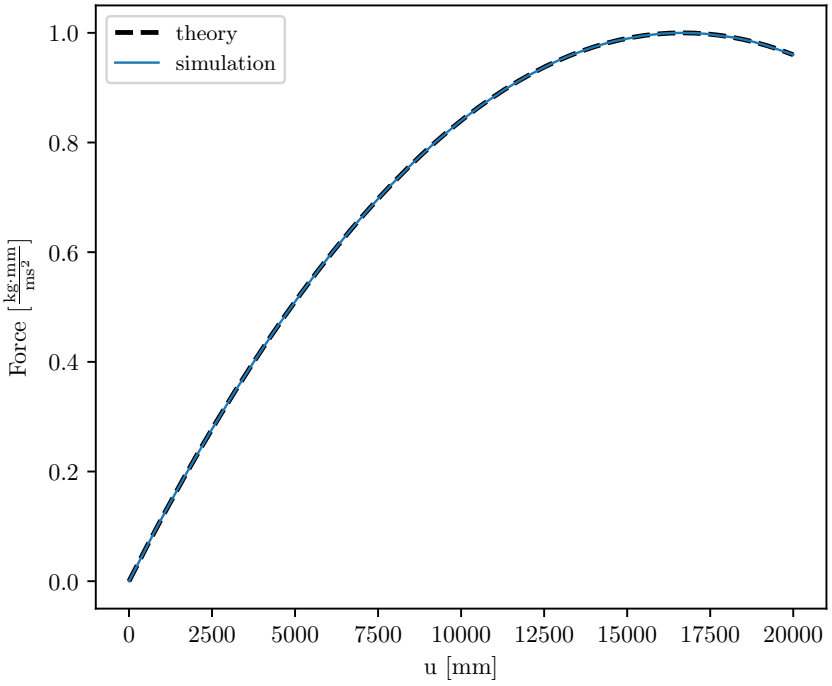


Figure 3.10 Extension of a non-linear spring under constant extension following equation (3.42). The smooth input function is perfectly reproduced by the simulation, even the decrease in the force while the spring is extending is modelled excellent.

Fig. 3.11 also shows the extension of a spring element but now the functional shape is taken to be discontinuous, an abrupt change is incorporated at around $u = 5000$ mm and a sine wave is inputted to investigate if OpenRadioss is able to match an irregular function like this. The agreement between the input function and the simulation is almost perfectly, only at the abrupt change the lines slightly deviate because of the sampling of the function. This shows that OpenRadioss is good in reproducing arbitrary spring curves.

3.4 Spring used in UNDEX analysis

The default springs used in UNDEX analysis use the keyword `MAT_NONLINEAR_ELASTIC_DISCRETE_BEAM` or `MAT_067`. This keyword defines a non-linear elastic discrete beam with 6 decoupled DOF which also allows preloading of the spring. In OpenRadioss this keyword can be read using the LS-DYNA format. OpenRadioss then maps this keyword to `/MAT/LAW108/` or `/MAT/SPR_GENE`. `/MAT/LAW108` is a general spring material with 6 DOF and allows for non-linear stiffness, damping and

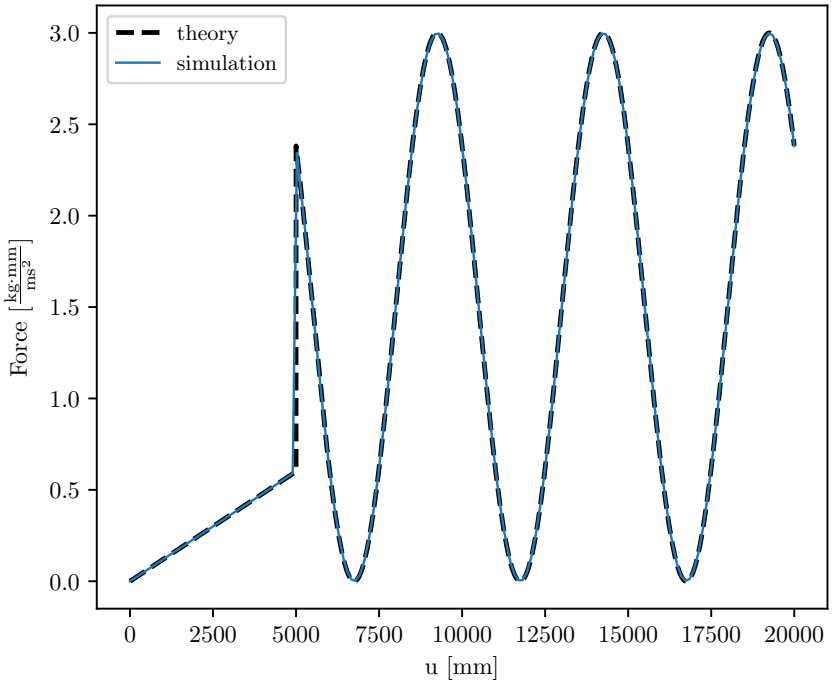


Figure 3.11 Extension of a non-linear spring under constant expansion. The irregular input function with an abrupt change is well reproduced by the simulation. There is a small disagreement at the displacement corresponding to the change of function, but this change is comparable to the sampling size of the input function.

different unloading scenarios. Additionally, `/MAT/SPR_GENE` allows to use deformation, force or energy based failure criteria for this element type (Altair, 2022a).

3.5 Conclusions

- The considered ideal spring performs extremely well for the force-displacement relation and the oscillation frequency.
- The considered damped ideal spring behaves similarly well as the undamped spring.
- Non-ideal springs do not produce the expected frequency for time evolution because the centre of the potential is not consistent between the theory and the simulation. Their force-displacement relation agrees perfectly with any input function. For the frequency of the oscillation, we have to do simulations that agree between the theory and simulation.

4 Compiling the code

This chapter focuses on how to compile OPENRADIOSS on your system. If the reader is not interested in compiling the code, the reader is referred to the next chapter.

There are two ways of obtaining OPENRADIOSS, the most straightforward way is downloading the 'day-release' executable from the website and use this to run simulations. However, this works but is not ideal for the following reasons:

1. The code is compiled using the GNU FORTRAN compiler (https://gcc.gnu.org/fortran/).
2. The code is not optimised for the current architecture.

Personal experience of the author and scientific research (Colfax Research, 2017; Halbiniak et al., 2022) show that the intel compiler is always faster even on AMD architectures[1]. Secondly, OPENRADIOSS wants to release an executable that everyone can use. This means that general instruction sets are used that are common on even old processors from the 2000s. Instead, Intel recommends to use `-xHost`, this generates a program with the highest possible instruction set for the compilation host (see Intel developers, 2010, 2023).

This chapter is divided in three parts, firstly, general software and settings that are required to run the code. Secondly, how to compile the code to run with multiple threads using OpenMP. Thirdly, how to compile the code to run with MPI and OpenMP to run over different memory domains. Chapter 5 focuses on optimising the compilation of the code.

4.1 Getting the code, required software and settings

OPENRADIOSS contains several large files, these large files contain third-party packages and their proprietary code for Altair's h3d files. To download the code, you need to make sure that you first install git lfs (git Large File Storage). Secondly, To use git lfs you need to initialise it as:

```
git lfs install
```

Once done, you can download the code from the GitHub repository:

```
git clone https://github.com/OpenRadioss/OpenRadioss.git
```

This will download the most recent version of OPENRADIOSS with all the latest fixes and additions. Because the released versions of OPENRADIOSS change daily, you should keep well track of the current version used. the current version of the repository can be determined by doing:

```
cd OpenRadioss
git log
```

The last command shows an output like the following:

```
commit 8e2cb72123b36b13a1e83c238327324854816f23 (HEAD -> main,
    tag: latest-20230919, origin/main, origin/HEAD)
Author: servbotaltr <102742919+servbotaltr@users.noreply.
   github.com>
Date:   Tue Sep 19 03:47:29 2023 +0200
```

[1] Note that NOS only has computers and servers with Intel Xeon processors

```
    Update Headers at Tue Sep 19 03:47:29 CEST 2023

commit c4b2e73a4ec7d7973e417cf87eeec87c336d8cd1 (tag: latest
    -20230918)
Author: Thierry Schwoertzig <104987175+Schwoertzig@users.
    noreply.github.com>
Date:   Mon Sep 18 11:49:27 2023 +0200

    /ALE/GRID/MASSFLOW:openMP fix

commit 073e777f6426e4cea6b5665dc4b80c2b1b5820a3
...
...
```

This shows the version string after 'commit', so for this version of the code, the version string is `8e2cb72123b36b13a1e83c238327324854816f23`. If you have this string stored for your project, it is always possible to retrieve this version of the code. The additional tag, `latest-20230919` is daily released by OPENRADIOSS and they have a limit of 6 tags, so after 6 days this information is lost and this tag itself does not say much. The other information `main` and `origin/main` indicate that you are on the main branch, `origin/head` indicates that you are on the latest commit (HEAD) of the branch.

The next step is making sure the required software is present. Table 4.1 shows the required software (Wienholtz et al., 2023).

Table 4.1 Required software to run OPENRADIOSS

software	version number
git lfs	>=3.0.2
cmake	>=3.20.4
make	>= 4.2.1
perl	>= 5.26
python	>= 3.6.0
GFORTRAN	>= 11.x
g++	>=11.x
cpp	>=11.x
gcc	>=11.x
IFX	>= 2021

To run with this software on the TNO-NOS Linux servers Goldfish or Marlin it is required to create symbolic links for the version 12 of the GNU compilers, this can be done simply by:

```
mkdir ~/bin/
cd ~/bin/
ln -s /usr/bin/gfortran-12 gfortran
ln -s /usr/bin/cpp-12 cpp
ln -s /usr/bin/g++-12 g++
ln -s /usr/bin/gcc-12 gcc
```

This is required because otherwise the GNU-7 versions of the compilers are used. If it is desired to use the intel compiler the following needs to be added to the run

command (rc) file[2] on Goldfish:

```
alias intel='source /uhome/tuitmanjt/intel/oneapi/setvars.sh
    intel64'
```

and for Marlin:

```
alias intel='source /Applications/oneapi2023/setvars.sh
    intel64'
```

After adding this to your rc file, you can run

```
intel
```

which will result in the following output and the initialisation of the intel compiler

```
:: initializing oneAPI environment ...
   -bash: BASH_VERSION = 4.4.23(1)-release
   args: Using "$@" for setvars.sh arguments: intel64
:: advisor -- latest
:: ccl -- latest
:: compiler -- latest
:: dal -- latest
:: debugger -- latest
:: dev-utilities -- latest
:: dnnl -- latest
:: dpcpp-ct -- latest
:: dpl -- latest
:: ipp -- latest
:: ippcp -- latest
:: ipp -- latest
:: mkl -- latest
:: mpi -- latest
:: tbb -- latest
:: vpl -- latest
:: vtune -- latest
:: oneAPI environment initialized ::
```

4.2 Single node and with OpenMP

The next step is compiling the starter. This can be done by:

```
cd OpenRadioss
cd starter
./build_script.sh -arch=linux64_gf
```

This compiles the starter and results in an executable in `OpenRadioss/exec`. Similarly, the engine can be compiled using GFORTRAN by

```
cd OpenRadioss
cd engine
./build_script.sh -arch=linux64_gf
```

[2] The rc file depends on which shell environment is used, .BASHRC for BASH, .ZSHRC for ZSH, .CSHRC for CSH, etc.

It is also possible to compile the engine using the intel compiler using

```
./build_script.sh -arch=linux64_intel
```

In OPENRADIOSS the compilation flags are accessible by the user, so it is possible to modify the compiler flags for better performance. This is possible to do with `OpenRadioss/engine/CMake_Compilers/cmake_linux64_intel.txt`, this file indicates the flags that are used in the code. Specifically, it is possible to modify lines 77 and 78. For some of the simulations at NOS, `-axSSE3,COMMON-AVX512 -no-fma -O3` was replaced by `-no-fma -Ofast -xHost -static`. More details about changing this can be found in Chapter 5.

Lastly, before running the code you are left with two executables, the starter and the engine. Before running simulations you need to initialise a few variables in your `.bashrc` file:

```
export OPENRADIOSS_PATH=/path/to/OpenRadioss
export RAD_CFG_PATH=$OPENRADIOSS_PATH/hm_cfg_files
export LD_LIBRARY_PATH=$OPENRADIOSS_PATH/extlib/hm_reader/
    linux64/:$OPENRADIOSS_PATH/extlib/h3d/lib/linux64/:
    $LD_LIBRARY_PATH
```

Running a simulation is performed in two steps, namely, the preparation of the simulation using the starter:

```
./dir_to_exec/starter_linux64_gf -i filename_0000.rad
```

followed by performing the actual simulation on N openMP threads using an engine executable

```
./dir_to_exec/engine_linux64_intel -nt N -i filename_0001.rad
```

or

```
./dir_to_exec/engine_linux64_gf -nt N -i filename_0001.rad
```

4.3 Run the code over MPI

On Goldfish and Marlin, the compilation of the code is very similar to the non-MPI version of the code. The starter does not need to be recompiled. However, the command to compile the engine is slightly different, to compile the code you will need to use:

```
./build_script.sh -arch=linux64_intel -mpi=impi
```

where the last keyword of command specifies which type of mpi is used (intel mpi here).

Importantly, you will need to indicate the number of threads per MPI rank as:

```
export OMP_NUM_THREADS=<Nthreads>
```

The next step is running the code, importantly, when running the code over MPI you should use the following command for the starter:

```
/directory/to/starter_linux64_gf -np <N> -i input_file_0000.
    rad
```

where <N> indicates the amount of MPI ranks that you want to run on. This is followed by a corresponding run that is performed on the same number of MPI ranks. Using the command

```
mpirun -np <N> --map-by socket:PE=<Nthread> --bind-to core /
    directory/to/engine_linux64_intel_impi -i filename_0001.
    rad
```

where <Nthread> is the number of threads per MPI rank. If the engine is run with the wrong number of MPI ranks this will produce an error.

4.4 Converting the output

OPENRADIOSS has three types of output. The output files are not directly readable by any standard free tool. Therefore, it is required to compile the converters for these files. OPENRADIOSS has three types of outputs. The first type is *.H3D files. In general, use of this file type requires the use of HYPERVIEW. Because of this the focus of this report is not on using this file type because it requires a commercial license. Instead, the focus is on the ANIM files or *A* files. *.H3D and ANIM files contain the same information, only the ANIM files can be converted to *.VTK files, which can be opened with PARAVIEW. In order, to get the ANIM to *.VTK converter, go to TOOLS/ANIM_TO_VTK/LINUX64 and execute

```
./build.bash
```

This creates the ANIM converter in your EXEC directory named ANIM_TO_VTK_LINUX64_GF. Now the ANIM files can be converted with:

```
./dir/exec/anim_to_vtk_linux64_gf filenameAnumber >
    filename_number.vtk
```

The third type of output are TH files or *T* files. These files store a smaller number of elements at a much higher frequency than the ANIM and *.H3D files. The time files are therefore ideal if you want to know in detail what is happening to many elements. The time files can be converted into a *.CSV file. To get the TH files to *.CSV converter go to TOOLS/TH_TO_CSV/LINUX64 and execute

```
./build.bash
```

This creates the TH files to *.CSV converter in your EXEC directory named TH_TO_CSV-_LINUX64_GF. Converting TH files can be done as:

```
./dir/exec/th_to_csv_linux64_gf filenameT01
```

4.5 Wrapper functions and aliases

This section focuses on wrapper functions and aliases in your rc files that will make working with OPENRADIOSS easier and more efficient.

4.5.1 Wrappers for converters

It is possible to create aliases in your .bashrc to quickly call the converter routines, this can be done as:

```
alias anim_to_vtk = /dir/exec/anim_to_vtk_linux64_gf
alias th_to_csv =/dir/exec/th_to_csv_linux64_gf
```

However, using these commands is still cumbersome if you want to convert many files at the same time. To prevent this, you can combine converting to *.VTK files into a single function as:

```
anim_to_vtk ='/dir/to/OpenRadioss/exec/anim_to_vtk_linux64_gf'
anim_to_vtk_all () {
  number_of_files =$(ls -lR $1A* | wc -l)
  maximum_index =$(( number_of_files ))
  start_index =1
  for (( i= $start_index ; i<= $maximum_index ; i++ ))
  do
    printf -v output_i "%03d" $i
    $anim_to_vtk $1A$output_i > $1_$output_i .vtk
  done
}
```

Using this function, you can convert all ANIM files in one go as:

```
anim_to_vtk_all filename
```

where FILENAME is the filename without the extension. The next step is combining this in a single function such that the TH files are also converted. This can be done by adding:

```
alias th_to_csv= /dir/to/OpenRadioss/exec/th_to_csv_linux64_gf
convert_radioss () {
  anim_to_vtk_all $1
  th_to_csv $1T01
}
```

Now, files can be converted as

```
convert_radioss filename
```

Overall, to combine all the functionality in your .BASHRC you can simply place the following text in it:

```
# ANIM converter
alias anim_to_vtk = /dir/exec/anim_to_vtk_linux64_gf

anim_to_vtk ='/dir/to/OpenRadioss/exec/anim_to_vtk_linux64_gf'

anim_to_vtk_all () {
  number_of_files =$(ls -lR $1A* | wc -l)
  maximum_index =$(( number_of_files ))
  start_index =1
  for (( i= $start_index ; i<= $maximum_index ; i++ ))
  do
    printf -v output_i "%03d" $i
    $anim_to_vtk $1A$output_i > $1_$output_i .vtk
  done
```

```
}

# Converter for all file types
alias th_to_csv= /dir/to/OpenRadioss/exec/th_to_csv_linux64_gf

th_to_csv= '/dir/to/OpenRadioss/exec/th_to_csv_linux64_gf'

convert_radioss () {
  anim_to_vtk_all $1
  th_to_csv $1T01
}
```

4.5.2 SMP version

For running all the processes of OPENRADIOSS in one go, it is most practically to create a function, this can be done by adding:

```
radioss () {
  radioss_starter -i $1_0000.rad
  radioss_engine -i $1_0001.rad -nt $2
  anim_to_vtk_all $1
  th_to_csv $1T01
}
```

Now you can run OPENRADIOSS on 4 cores as:

```
radioss filename 4
```

4.5.3 MPI version

For MPI the code is slightly different, given by:

```
radioss_mpi () {
  export OMP_NUM_THREADS = $2
  radioss_starter -np $3 -i $1_0000.rad
  mpirun -np $3 --map -by socket:PE=$2 --bind -to core /dir/to
     /exec/engine_linux64_intel_impi -i $1_0001.rad
  anim_to_vtk_all $1
  th_to_csv $1T01
}
```

Now you can run radios on 16 cores with 8 MPI ranks and 2 threads per MPI rank as:

```
radioss_mpi filename 2 8
```

The recommendation for the MPI version is to use only a single thread per MPI rank.

4.5.4 .BASHRC file

Based on the different components above the general recommendation is to add the following lines of code to the .BASHRC file of users:

```
# aliases to radioss
radioss_starter = /dir/to/exec/starter_linux64_gf
radioss_engine = /dir/to/exec/engine_linux64_intel
```

```
# ANIM converter
alias anim_to_vtk = /dir/exec/anim_to_vtk_linux64_gf

anim_to_vtk ='/dir/to/OpenRadioss/exec/anim_to_vtk_linux64_gf'

anim_to_vtk_all () {
  number_of_files =$(ls -lR $1A* | wc -l)
  maximum_index =$(( number_of_files ))
  start_index =1
  for (( i= $start_index ; i<= $maximum_index ; i++ ))
  do
    printf -v output_i "%03d" $i
    $anim_to_vtk $1A$output_i > $1_$output_i .vtk
  done
}

# Converter for all file types
alias th_to_csv= /dir/to/OpenRadioss/exec/th_to_csv_linux64_gf

th_to_csv= '/dir/to/OpenRadioss/exec/th_to_csv_linux64_gf'

convert_radioss () {
  anim_to_vtk_all $1
  th_to_csv $1T01
}

# Define the radioss command
radioss () {
  radioss_starter -i $1_0000.rad
  radioss_engine -i $1_0001.rad -nt $2
  anim_to_vtk_all $1
  th_to_csv $1T01
}

# Define the radioss MPI command
radioss_mpi () {
  export OMP_NUM_THREADS = $2
  radioss_starter -np $3 -i $1_0000.rad
  mpirun -np $3 --map -by socket:PE=$2 --bind -to core /dir/to
     /exec/engine_linux64_intel_impi -i $1_0001.rad
  anim_to_vtk_all $1
  th_to_csv $1T01
}
```

5 Computational performance

This chapter focuses on the performance of OPENRADIOSS and how well it scales with number of CPUs and number of resolution elements, a set of benchmark simulations are performed to assess the computational performance of OPENRADIOSS. For the performance analysis the cantilever beam simulation presented in Chapter 2 is taken as a benchmark.

5.1 Performance of weak scaling test

Here we perform a weak scaling test, here the problem is not kept identical, but the problem has more resolution elements. For our cantilever benchmark, this means that the resolution elements are smaller and correspondingly, the time steps are smaller. This means that the problem is harder to solve the more resolution elements are present. The simulations at different resolutions are all performed on the same computing nodes. The problem is not identical, therefore it is called a weak scaling test instead of a strong scaling test (the definition of Potter et al., 2017, is followed). Despite this it will give a good idea of how well the performance of the code is when used on bigger problems.

When weak scaling a problem, this means that the time step is decreased by the same factor as the resolution because $\Delta t \propto 1/\Delta x$, therefore the number of calculations scales as $1/\Delta x$. Additionally, for beam elements when increasing the spatial resolution, this also means that the amount of resolution elements increases with the same amount and depends on $\propto 1/\Delta x$. For shell elements the situation is different because there are also new elements appearing at the sides of shell elements and therefore the dependence on the resolution is $\propto 1/\Delta x^2$. For brick elements (3D) the dependence scales even stronger with the resolution as $\propto 1/\Delta x^3$ (brick elements are not considered in this report[1]). This means the following scaling relations are found for beam, shell, and brick elements:

$$t_{\text{wallclock,beam}} \propto \frac{1}{\Delta x^2}, \tag{5.1}$$

$$t_{\text{wallclock,shell}} \propto \frac{1}{\Delta x^3}, \tag{5.2}$$

$$t_{\text{wallclock,brick}} \propto \frac{1}{\Delta x^4}. \tag{5.3}$$

However, this is not the whole story, to link this to the number of resolution elements, this needs to be connected with the number of resolution elements. In the case of beam, shell, and brick elements the scaling with resolution mass goes as $M_{\text{res}} \propto \Delta x$, $M_{\text{res}} \propto \Delta x^2$, and $M_{\text{res}} \propto \Delta x^3$ respectively. Furthermore, M_{res} directly correlates with the amount of resolution elements as $N \propto 1/M_{\text{res}}$. This means that for the beam,

[1] Brick elements are not common in FEM models of ships and submarines, these consist mainly out of shell and beam elements. Sometimes engines or big structures are modelled as brick elements, but these are often not important for the structural response.

shell, and brick elements we have:

$$t_{\text{wallclock,beam}} \propto \frac{1}{M_{\text{res}}^2} \propto N^2, \tag{5.4}$$

$$t_{\text{wallclock,shell}} \propto \frac{1}{M_{\text{res}}^{3/2}} \propto N^{3/2}, \tag{5.5}$$

$$t_{\text{wallclock,brick}} \propto \frac{1}{M_{\text{res}}^{4/3}} \propto N^{4/3}. \tag{5.6}$$

This result might first seem counter intuitive but keep in mind that in the case of N elements the resolution size for beam elements is much smaller than for shell and brick. This is because the refinement is done in fewer dimensions. So what does it mean for models if they are below equations (5.4)-(5.6)? This means that the wall clock time is not dominated by solving the equations but rather is dominated by the overhead[2] of running OpenRadioss. If the scaling is steeper than equations (5.4)-(5.6), it means that the implementation of the physics can be made more efficient.

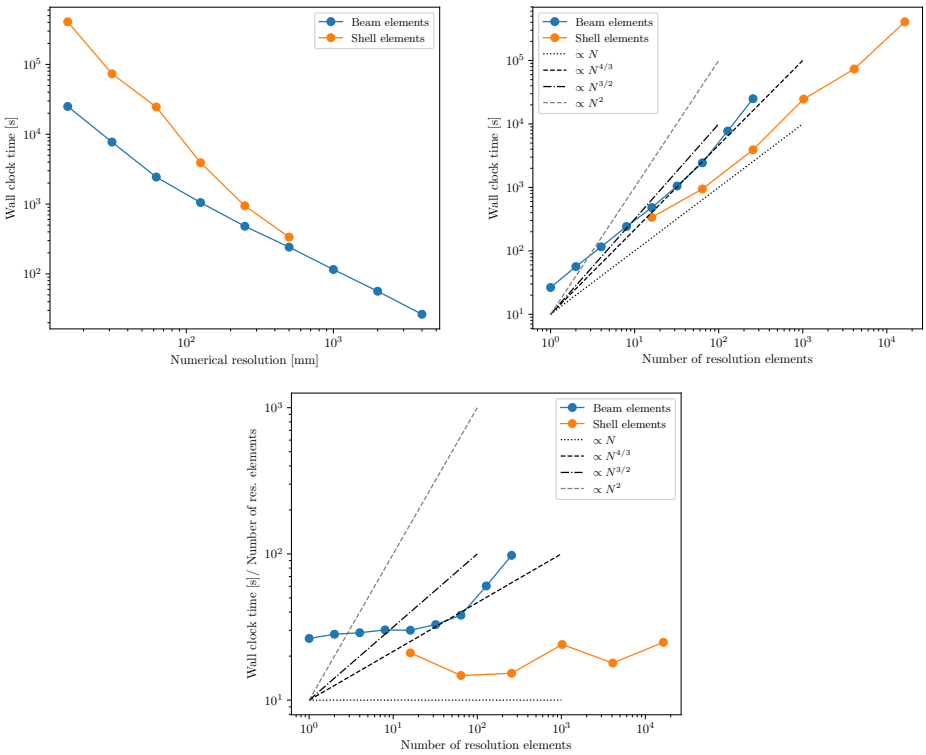


Figure 5.1 Different measures of the computational performance of OpenRadioss for simulations with shells (orange) and beams (blue). We find that the performance of the shell elements follows quite closely a scaling of $\propto N$, which is weaker than theoretically would be expected. Similarly, the beam elements scale close to $\propto N$ for low number of elements while for larger number of elements the scaling approaches more $\propto N^2$ which is as expected.

In Fig. 5.1 we show how well OpenRadioss scales its wall clock time as a function of resolution elements. The two highest resolution shell element simulations were performed on 4 and 6 cores respectively and the wall clock time was corrected for this[3]. For beam elements the scaling first is linearly, less steep than theoretically

[2] Overhead is the extra indirect computation time required to perform the calculation that do not include the main computation.

[3] Corrected as in including the wall clock time of all 4 or 6 cores.

expected. At around 10^2 resolution elements it starts to scale extremely close to N^2. This indicates that the scaling of beam elements behaves as expected. For shell elements the dependence on the number of resolution elements scales almost linearly, while it is expected to scale as $N^{3/2}$. This indicates that the scaling of shell elements scales a lot less strong than expected, even at 10^4 resolution elements the scaling is still weak with the number of resolution elements. This might be indicating that the simulation time is dominated by something else than the calculation of the EoM of the shell elements.

5.2 Performance of strong scaling test (SMP)

This section focuses more on strong scaling tests, following Potter et al. (2017) and Schaller et al. (2016). A strong scaling test is calculating a fixed problem with a different number of CPUs or threads. This section focuses on multiprocessing using shared-memory multiprocessing (SMP). This means that the same memory is accessed by multiple threads. To get an impression of the performance, simulations with different number of threads are compared to each other. The GNU and intel compilers are both investigated. The highest resolution simulation of the cantilever beam is used (see chapter 2) for a reduced time such that the simulation on one core can be completed within 2 hours. In terms of computing time there are two important times, firstly the wall clock time is the time it takes the simulation to run from the start time to the end time. The wall clock time therefore is the total time that is measured with a stopwatch while waiting for the simulation to complete[4]. The second time is the CPU time, this is the total amount of time CPUs have used to perform the simulation. This means that if multiple CPU work 100 per cent of the time for a certain wall clock time, the total CPU time is the amount of cores times the wall clock time. OPENRADIOSS does not explicitly log down the amount of CPU time that is used, because of this the ideal CPU time is used which is the wall clock time times the number of threads used. Fig. 5.2 shows the total CPU time and wall clock time as a function of number of CPU cores. The dotted line shows the ideal scaling. Fig. 5.2 shows that up to 4 cores the scaling is nearly perfect, for more cores the wall clock time is flattening, this indicates that the extra computational resources (i.e. extra cores/threads) are not performing more work in the same amount of times but rather the work to parallise the problem becomes the dominant work performed by the program. This implies that using 16 or more threads for this problem is not efficient. Similarly, the right shows that up to 4 cores the total CPU time is almost identical, this indicates that the code scales very well up to this point. Beyond this the code takes progressively more CPU time. When the performance between the Intel and GNU compiler is checked it can be noted that they are almost performing the same, GNU compiler is a bit slower than the Intel compiler, but the difference is very small and sometimes the GNU compiler is actually quicker.

The results of Fig. 5.2 are very focused on the test case of the cantilever beam. To make the test applicable to more general problems, Fig. 5.3 shows the amount of shell elements per core. Based on this 4×10^3 shell elements per core still give excellent convergence, furthermore, using 10^3 or less shell elements per core does not improve the performance.

To improve the performance of compiled code it is possible to use compilation flags that optimise certain parts of the code. The only obvious drawback is that simulations

[4] OPENRADIOSS measures this every time a simulation is performed.

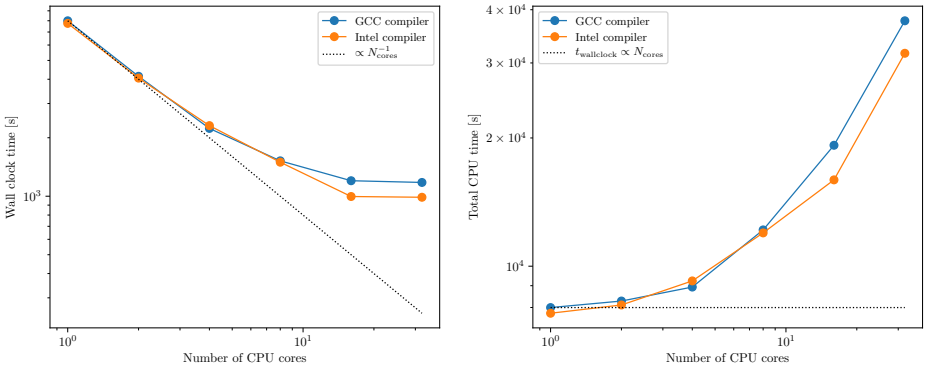


Figure 5.2 Comparison of the wall clock time (left) and CPU time (right) for the strong scaling test using the GNU and Intel compiler (different colours). The dotted line indicates the ideal scaling of the problem. We find that the code scales very well up to 4 cores for the cantilever beam problem.

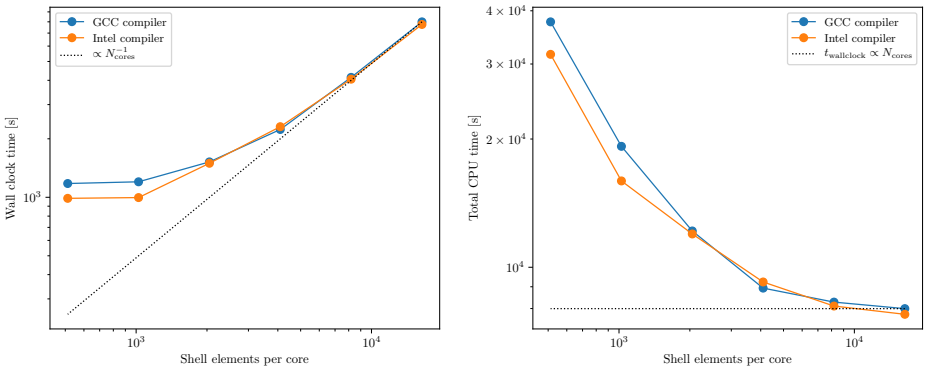


Figure 5.3 Comparison of the wall clock time (left) and CPU time (right) for the strong scaling test using the GNU and Intel compiler (different colours), instead of number of CPU cores, the x-axis shows the number of shell elements per core. The dotted line indicates the ideal scaling of the problem. The problem scales very well up to around 4×10^3 shell elements per core.

with more improved compilation flags take longer to compile. Common flags are (IT Boston University, 2023; Intel developers, 2023):

- -O2, for more extensive optimisation.
- -O3, more aggressive optimisation with longer compilation times, especially recommended when the code contains loops that involve intensive floating-point calculations.
- -OFAST, same as -O3 with -NO-PREC-DIV and -FP-MODEL=FAST 2[5]. The first extra compilation flag allows divisions to be calculated as multiplications with the reciprocals and the second flag optimises the floating-point data more.
- -XHOST, optimises the code for use on the specific type of CPU that is used.
- -IPO Optimisation that checks if during the compilation more common functions can be found such that the code can be optimised even more.
- -STATIC can improve the start time and makes sure the code runs in limited environments because it contains all the necessary libraries. The libraries are not dynamically linked in this case.
- -FAST, this is a shortcut for -OFAST -IPO -STATIC -XHOST.

[5] Normally, you want to avoid using -FP-MODEL=FAST 2 because it will change the accuracy and rounding off of numbers, this can cause many problems in libraries like BLAS or LAPACK.

Of all these compiler flags, the -IPO flag does not work with OPENRADIOSS, therefore, also the -FAST flag does not work. The other flags, they all work out of the box with OPENRADIOSS. Fig. 5.4 shows the CPU time for the different compiler options. This shows that -OFAST reduces the CPU time by a bit more than 10 per cent while the inclusion of -XHOST results in a performance improvement of 25 per cent. Not using the -AXSSE3,COMMON-AVX512 flag results in a slightly better performance of another 5 per cent. Combining these changes results almost in a reduction with a factor of two in the total computing time.

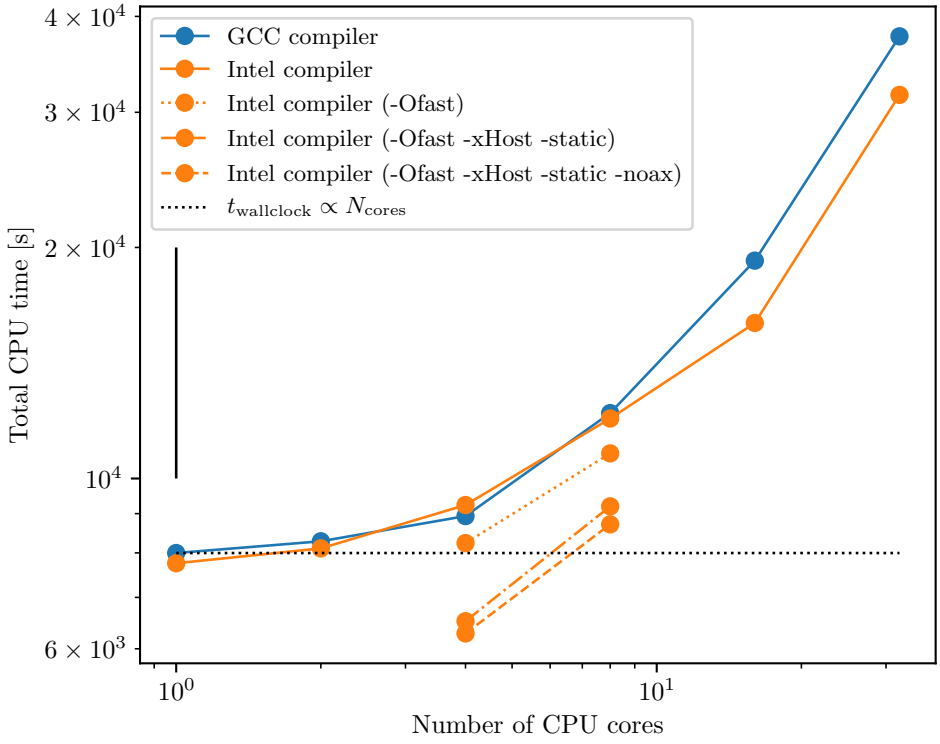


Figure 5.4 Comparison of the CPU time for the strong scaling test using the GNU and Intel compiler (different colours) and compiler flags (line styles) as a function of number of CPU cores. The black dotted line indicates the ideal scaling of the problem. Using different compilation flags for OPENRADIOSS improves the speed with a factor of two.

5.3 Performance of strong scaling test (MPP)

§ 5.2 focused on the SMP version of OPENRADIOSS, here the focus is on the massive parallel processing (MPP) version in combination with SMP. This version is recommended to use because it is the fastest version (personal communication Lequiniou, 2023). The implementation of MPP is based on the messaging passing interface (MPI). MPI is a standardised communication protocol used in parallel computing to enable communication and coordination between processes in a distributed system over multiple non-uniform memory access (NUMA)[6] regions or nodes that is used to implement MPP. MPI allows for efficient data exchange and synchronisations among multiple NUMA regions or nodes. In the case of using MPI it is also possible to use a combination of MPI and openMP. This means that each node uses multithreading on multiple cores at the same time. It depends on the application itself what combination

[6] For information see this link.

of MPI vs SMP is most efficient. In general, for ideal programs it is fastest to use one MPI rank per NUMA region and use OpenMP for multithreading on the cores in the NUMA region. This is expected to be fastest because it reduces the MPI communication to only different NUMA regions and nodes and has the lowest memory usage inside a NUMA region because elements are not duplicated in memory[7].

Additionally, an investigation of the different compiler flags is done similarly to § 5.2. The compiler flags are systematically changed to investigate if the compiler flags have impact on the performance of the code.

Fig. 5.5 shows the computing time of the cantilever beam benchmark as a function of number of cores. The different line styles indicate the number of threads per MPI rank and the colour indicates the compiler flags, all as a function of CPU cores. The lowest line of the bottom figure corresponds to 1 thread per MPI rank, this means that when 16 or 32 cores are used there are 16 or 32 MPI ranks running, respectively. Some important conclusions from this are as follows:

- Running with as many MPI ranks as possible results in the shortest simulation time. However, this conclusion might not hold for 32 cores (every time), but this might be related to the fact that the problem is too small for 32 cores or unknown to us, someone else used the computer cluster while performing these simulations.
- When using MPI the compiler flags have barely an impact on the computing time.

The results are interesting, but the result is surprising that changing the compiler flags in this case does not influence the performance.

Bini Leite et al. (2021) investigated multiprocessing of OpenRadioss on many cores for a large model. Fig. 5.6 shows the result of a strong scaling test for a large ditching model. Their ditching model includes a FEM model for an aeroplane, a SPH mesh for the water and Lagrangian solid elements (brick elements) for the water mesh that is not close to the crashing plane. Bini Leite et al. (2021) shows that multi-domain simulations with different time stepping sizes per domain region can reduce the computing time by a factor of 3 for large simulations. They also show that the mono-domain simulation shows almost perfect scaling up to 150 cores and only at almost 300 cores performs slightly less good but is still only 12 per cent off ideal scaling. The scaling of the multi-domain simulation is less close to ideal scaling because there is most likely more overhead. Despite this weaker scaling, the code takes a much shorter time to complete for the multi-domain simulation. Note that performing a multi-domain simulation requires a larger amount of preparations by the user because the multi-domain needs to be constructed correctly and this often requires verification that the used multi-domain approach works. For large simulations, the multi-domain approach might be worth the effort given that it also reduces the computation time. It is noted that within TNO-NOS, currently also 3dcav simulations are performed with a multi-domain approach for the fluid mesh using exMesh (see Tuitman, 2023).

5.4 High-performance computing options

In this section some advanced options of OpenRadioss are discussed that are of interest at the TNO-NOS Linux servers and when using the queueing system on the high-performance computing cluster of TNO.

[7] For MPI communication neighbouring elements that are around the region that is calculated on an MPI rank are required to be in memory for each MPI rank, this means that all the border elements are duplicated in memory.

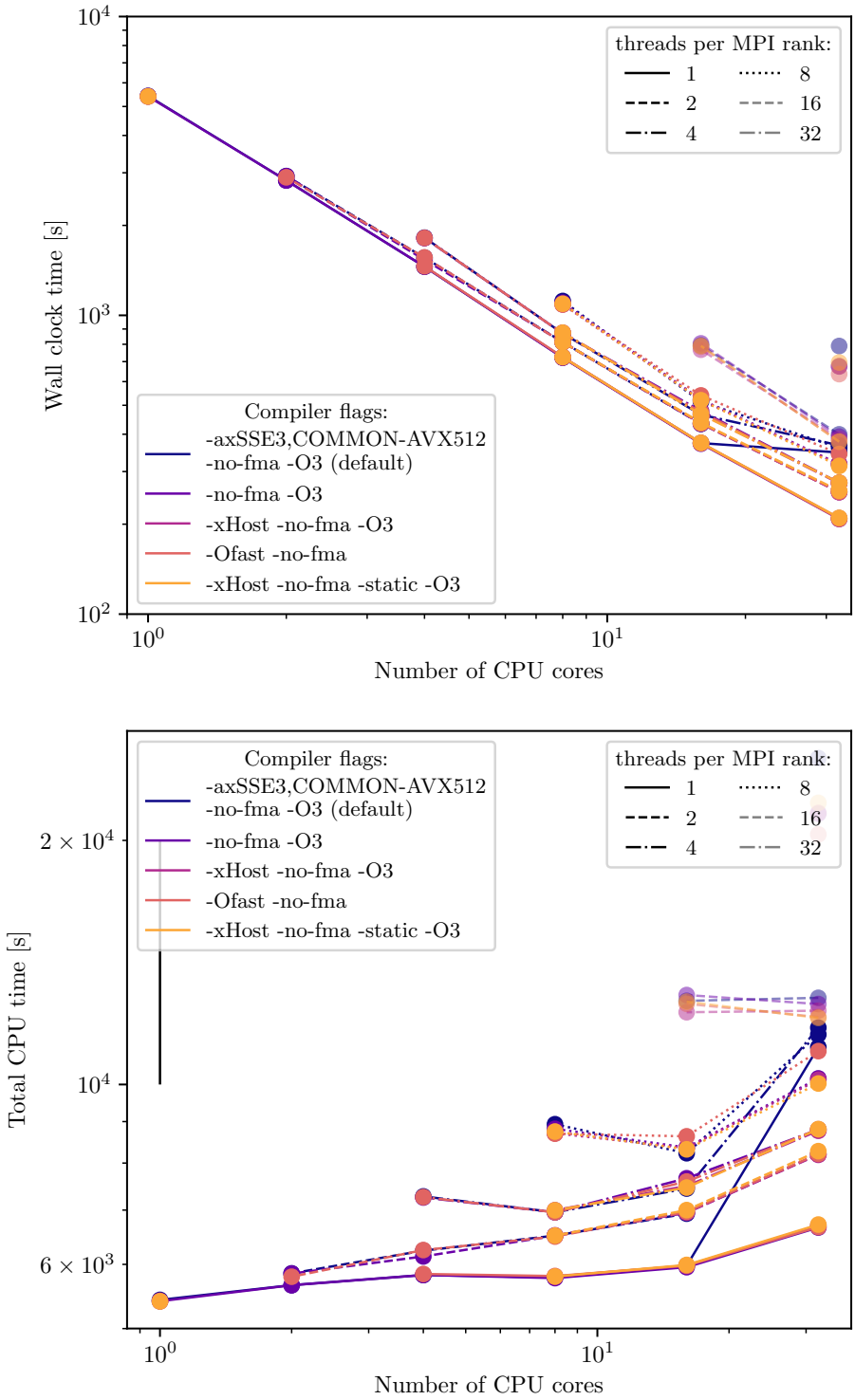


Figure 5.5 Comparison of the wall clock time (top) and CPU time (bottom) as a function of number of cores for the strong scaling test using the different Intel compiler flags (different colours) and different number of MPI ranks (different line styles). Running with as many MPI ranks as possible gives the shortest runtime (lowest line in the bottom), using fewer MPI ranks increases the total CPU time (e.g. top dashed transparent line). The compiler flags have limited effect on the performance.

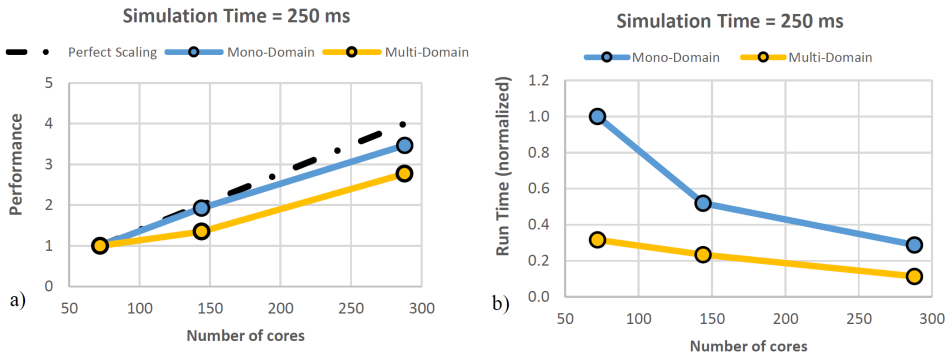


Figure 5.6 Strong scaling performance comparison for a large dishing model – mono-domain (blue) vs multi-domain (yellow). Ideal scaling is shown by the black dot-dashed line a) Linear speed up versus number of cores. b) Run time versus number of cores. Multiprocessing is more efficient for mono-domain simulations, multi-domain simulations still reduce the computing time significantly. Multi-domain simulations are also much more computational efficient because they take only a third of the time of a mono-domain simulation. Figure taken from Bini Leite et al. (2021).

5.4.1 Control File

The control file is an optional file that can be created while a run is ongoing. The control file is named as *.CFL with the same prefix as the parameter file for the engine (see Altair, 2023, for details). This file can specify to print more information or to make more frequent outputs of the ongoing simulation. The main reasons to use a control file are:

- Stop the simulation directly, or at a specified time or cycle number.
- Create an animation directly, or from a specified time or cycle number.
- Retrieve more information of the current simulation like the simulation time, time step, cycle, energy information and estimated remaining time.

In general, when the control file is created OPENRADIOSS will read the file as quickly as possible and execute the instructions in the file. The control file has two options to stop the simulation, the first is /KILL which stops the simulation and /STOP which stops the program and creates a restart file. In general, it is preferred to use /STOP because it creates a restart file. The control file is directly executed, except if either the /TIME/TIMEVALUE or /CYCLE/CYCLENUMBER are written in the file with a corresponding time value or cycle number when to terminate. Other comments are /ANIM, /H3D, /RFILE, /INFO, /CHKPT which all create extra information during the termination, namely, ANIM or H3D files, restart file, extra information, and a check data file.

5.4.2 Multiple engine files

It is possible to have multiple engine files that modify the way in which the solution of the simulation is obtained (Altair, 2023). Example of variations are for example simulation time, output files, time stepping conditions or other numerical values. It is also possible to use different damping strengths or remove parts of the boundary conditions all while using the same starter file.

5.4.3 Checkpoint file

OPENRADIOSS allows the writing of a checkpoint file. This is a file that can be used to continue running the used simulation (Altair, 2023). In OPENRADIOSS checkpoint files

need to be created on the spot and currently no automatic way of creating checkpoint files is done. Commonly used HPC codes often have checkpoint files that can be specified in the input file itself, e.g. Gadget (Springel, 2005) or Swift (Schaller et al., 2023). This is extremely useful for a various of reasons. Firstly, if a code problem is found, the problem can always be simulated from the last checkpoint which typically are spanned between 3 to 6 hours in computing time. Secondly, using HPC facilities can be done automatically because the checkpoint files are automatically created, and this guarantees that simulations can be restarted with a resubmission script.

Currently, check-pointed simulations can be rerun using:

```
/dir/to/engine/engine_linux64_intel -np 4 -checkpoint
    runname_0000.rad
```

Overall, now it is not possible to use the full functionality of checkpoint files to perform large simulations. Because checkpoint files need to be created on demand and cannot be created in an automatic way.

6 Advantages and Disadvantages of OPENRADIOSS

In this chapter we summarise the different advantages and disadvantages of using OPENRADIOSS as compared to commercial explicit solvers like LS-DYNA.

6.1 Advantages

- Open source:

  OPENRADIOSS is open-source software under the GNU Affero General Public License (GNU AGPL) license. Using Open-source software has a large amount of advantages compared to commercial packages like LS-DYNA, the following are some of the advantages:
  - There are no license costs for usage of OPENRADIOSS. This implies that there is no limit for the number of simulations performed with OPENRADIOSS, therefore, with OPENRADIOSS larger and/or more simulations can be performed as compared to LS-DYNA. This implies more detailed and/or more scenarios can be investigated in parallel. This is contrary to LS-DYNA, for which the license costs have increased and might increase more in the future.
  - Access to the source code:
    * It becomes much easier to keep libraries developed by TNO up to date (e.g. 3DCAV, SIT, etc.)
    * It becomes possible to get access to more variables and with modifications of the code it is even possible to get access to almost all variables.
    * It becomes possible to solve small bugs and propose code changes for this.
    * It is possible to collaborate with universities that want to develop material routines or other features in OPENRADIOSS.
  - Access to the latest developments of OPENRADIOSS:
    * The most recent version of OPENRADIOSS is available on GitHub and allows us to directly access a new version of the code once a bug has been solved.
    * Transparency of current bugs and problems in the code.
    * Transparency of what is currently being developed for future code releases of OPENRADIOSS.
  - It becomes possible to optimise the code for the CPUs of our servers, this can save a factor of 2 to 3 in the computing time.
- The support of OPENRADIOSS is outstanding on the GitHub and over e-mail. Contrary, the support of LS-DYNA has decreased significantly after Ansys took over LS-DYNA. Note that for OPENRADIOSS even small questions related to unclear documentation are quickly answered by the support of OPENRADIOSS or GitHub.
- There is an open-source community that works with OPENRADIOSS and that helps improve OPENRADIOSS faster than Altair can do on its own.
- For some functionality OPENRADIOSS has more general and flexible structure that allows the user to modify more variables.
- OPENRADIOSS is one of the few FEM packages (to our knowledge) that has an input and output unit system. This means that in OPENRADIOSS it is possible

to do simulations in different unit systems and correspondingly have the models (e.g. material models) with correctly converted units. One key feature of OPENRADIOSS is that there is an input unit system and an output unit system that is set by the user, this allows the user to use an input model in ms, g and cm and have the output in SI, without needing to convert the input file of the FEM model.

- OPENRADIOSS has a list of (example) problems available that allows a new user to try problems close to what they are intending to do. We note that this list is not complete and some problems that were published are not included. Also recently, OPENRADIOSS added all the papers on the website that use OPENRADIOSS.

6.2 Disadvantages

- The most obvious disadvantage is that many colleagues need to learn how to work with OPENRADIOSS, this implies that colleagues need to spend a significant time on this.
- The team of OPENRADIOSS recently started working in an open-source way. They tend to approach the issue system (a ticket system on GitHub) in a bit of an unorthodox approach. Rather than replying directly to the issue (ticket) they quickly send an email to the reporter (person reporting the problem). Then they try to solve the problem over email or even an online meeting. This has the drawback that not all OpenRadioss users can be aware of the ongoing issue. This shows clearly that they want to be engaged with the users, however, this makes the status of ongoing problems unclear for the community of other users. This also indirectly impacts TNO because open issues of features that we (intent to) use do not contain (much) information of the status of ongoing problems.
- OPENRADIOSS does not use a proper version management for their release versions. Rather, they daily release a new version of the code and keep only 6 versions of the code available for download (last 6 working days). This is highly undesirable because it is unclear when certain features are implemented into the code. Furthermore, it is harder to keep track of different versions of the code. Also having a release of the code of the last 6 days seems unnecessary because the last version can always be downloaded with git. Rather it would be more beneficial for the users to have version releases when the number of features implemented / bugs fixed exceeds a certain number. This also allows the releases to include information about the new features and gives a much clearer version management. We note that OPENRADIOSS is exploring various strategies for improving version management, and their current approach involves researching different options to determine which will be most effective. Therefore, we expect this disadvantage to be resolved in the near future.
- OPENRADIOSS lacks some of the functionality of LS-DYNA. A good example of this is the shell elements. In OPENRADIOSS there are just three types of shell elements for shells with four nodes[1] while in LS-DYNA there is much more different shell elements. It is noted that the Belytschko and Tsay (1983) shell element that TNO-NOS uses extensively is present in both OPENRADIOSS and LS-DYNA. Therefore, we do not expect this to be a problem for TNO-NOS. The lack of implementation of some features might extend to other functionality that is desired for a FEM package. Note that the Zeng and Combescure (1998) shell

[1] three types of shell elements and for one type there are four different penalisation methods for the hourglass.

elements are not present in LS-DYNA but are present in OPENRADIOSS.

6.3 Disadvantages OPENRADIOSS specific

The following disadvantages of OPENRADIOSS are only OPENRADIOSS specific and have no equivalent in for example LS-DYNA. Because OPENRADIOSS is open source a detailed assessment can be made of some OPENRADIOSS specific disadvantages.

- The quality of the code is not always good. For example, different pieces of code lack proper naming of variables. For example, in some pieces of code the variables names are e.g. called, A1, A2, A3, .. and B1, B2. In these cases, it is possible to determine what exactly is happening but it requires more work than in the case of a properly documented code. Furthermore, large parts of the code are not commented at all and because of that it is unclear what is happening to the specific variables or what is happening in the different files. Also, the names of the different FORTRAN files have obscure names which do not make the context of the different files directly clear. Some recent effort has been made in using the free FORTRAN90 format for all new developments. However, not much effort is put into using a clear naming convention for OPENRADIOSS. We expect that when this is improved, contributing to OPENRADIOSS will be more easy.
- OPENRADIOSS made the choice to use IFX instead of IFORT. IFX is a next-generation compiler developed by INTEL. However, it contains still bugs and it does not support all the features of FORTRAN. Because of this INTEL itself does not recommend to use IFX but rather says that it still needs to convince the FORTRAN community that IFX is worth it (Green, 2022). Furthermore, IFX still performs less well than IFORT and this might require a couple of more years of work on INTEL's side (Green, 2023; Sait, 2021). Furthermore, some instructions for MPI are not implemented in IFX and this means slower performance over MPI as compared to using IFORT (IFX developers, 2023). Fixing these issues is work that needs to be resolved by INTEL and might take several years.
- Running the code with MPI and OpenMP results in the conclusion that running with as much MPI ranks as possible gives the best computational performance. This is not what normally is the case given that using MPI causes overhead. MPI causes overhead because it requires the different MPI ranks to communicate to each other, this makes the code slower. Additionally, information of neighbouring elements needs to be send to each other, this means that in an ideal program running with as many MPI ranks as possible is not the fastest solution. Given that for OPENRADIOSS this is is the fastest solution implies that there is some possible performance in the OpenMP part of the code.
- Parts of the manual are unclear, incomplete, or outdated. The manual has for example inconsistent use of the letter t which is used in the same equation both for time and thickness of the shell elements. In the theory manual there are sometimes steps not shown which confuse the reader. Some parts are also outdated like the best options to use for multiprocessing are completely different from what the developers of OPENRADIOSS recommend.
- OPENRADIOSS does not use a large set of verification problems using continuous integration and continuous deployment (CI/CD). CI/CD allows developers of the code to test the functionality of the program while developing the code. The main focus of the CI/CD of OPENRADIOSS is on checking if the code still compiles on Linux and Windows rather than also verifying if the solution remains correct.

7 Conclusions and Recommendations

For shell and beam elements OPENRADIOSS works well. Good performances for spring elements are found, but complicated springs need additional investigation. It is recommended that TNO-NOS continues investigating the applicability of OPENRADIOSS for use as an explicit finite element solver. In § 7.1 we discuss the conclusions in detail, §§ 7.2 and 7.3 discuss the recommendations for TNO and Altair.

7.1 Conclusions

Chapter 1 gives a short overview of the motivation of investigating OPENRADIOSS as a replacement for LS-DYNA. This is followed by a historical perspective on OPENRADIOSS, recent comparisons of OPENRADIOSS and LS-DYNA, and a summary of recent developments and use cases of OPENRADIOSS in the literature. We emphasise that the recent developments and use cases are publicly available.

Chapter 2 focuses on verifying shell and beam elements. A benchmark test following a cantilever beam is constructed. The frequency and equilibrium conditions for a cantilever beam are derived and used for a comparison with shell and beam elements in OPENRADIOSS. The different shell and beam formulations of OPENRADIOSS are explained and the strengths and weaknesses of the different shell element formulations are explained. It is found that the Belytschko and Tsay (1983) formulation does not create converging results and match the analytical solution of the cantilever beam within 7 per cent (depending on the resolution less, down to less than 2 per cent, but without convergence). The Batoz and Dhatt (1990) Q4γ24 shell element also has trouble reproducing the correct solution due to locking but clearly shows convergence to a slightly wrong solution. The Zeng and Combescure (1998) shell elements are found to produce the best behaviour and agree well (within 5 per cent) with the analytical solution of the cantilever beam. For stability of time integration it is recommended to use a slightly more strict CFL condition of $C_{\mathrm{CFL}} \leq 0.75$ than the standard in OPENRADIOSS which corresponds to $C_{\mathrm{CFL}} = 0.9$ when using a regular mesh. We expect that $C_{\mathrm{CFL}} = 0.9$ is fine for an irregular mesh where the smallest time steps are caused by a few small resolution elements.

Chapter 3 constructs a simple spring model which consists out of two nodes and a single spring element with a mass, a force is placed on one of the nodes such that the spring starts oscillating, or a constant force is imposed while measuring the required force to extend the spring. Using this model, the spring models of OPENRADIOSS is tested for ideal and non-ideal springs. For ideal springs the force-displacement relation and the frequency of the oscillation are as expected. For non-linear springs with monotonically increasing spring energies the theoretical frequency is derived and used to compare with the numerical simulations, however, not a one-to-one comparison was made, which caused a discrepancy between the theory and simulation[1]. The force-displacement relation is reproduced for arbitrary input functions, even functions that show a decreasing force as the spring is extended can be reduced perfectly.

Chapter 4 explains how to compile OPENRADIOSS on the servers used by TNO NOS, this is done for both the shared-memory multiprocessing (SMP) and message passing interface (MPI) version of OPENRADIOSS. A recommendation of functions to add

[1] This is because we imposed a force on the spring but this caused the spring to oscillate in only one side of the potential, this means that we did not a one-to-one comparison. The comparison should be performed on an already extended spring that oscillates instead of using a simulation of a non-extended spring with an imposed force

to the .BASHRC file for OPENRADIOSS users is given such that OPENRADIOSS simulations can be performed as efficiently as possible. This way simulations can be performed and converted in a single command.

Chapter 5 investigates strong and weak scaling tests for OPENRADIOSS. The weak scaling test indicates that the code scales very well for larger number of shell and beam elements. The strong scaling test on the SMP version of OPENRADIOSS indicates that 4×10^3 elements per core leads to excellent scaling while more elements per core slows down the code more. Furthermore, different compiler flags are compared and using a compiler flag that uses -XHOST and/or -OFAST results in a better speed performance of around a factor of two. When using the MPI version of OPENRADIOSS the results are different, the compiler flags do not seem to influence the total CPU time much. For the MPI version of OPENRADIOSS the shortest simulation time can be achieved by using as many MPI ranks as possible and only use a single thread per MPI rank.

Chapter 6 summarises the advantages and disadvantages of OPENRADIOSS. Overall, the advantages of OPENRADIOSS outweight the disadvantages.

7.2 Recommendations for TNO

It is recommended that TNO-NOS continuous investigating the applicability of OPENRADIOSS for use as an explicit finite element solver. For shell and beam elements OPENRADIOSS has been working well. Recommendations for future improvements are:

- Investigate further non-linear spring elements focused on their frequency.
- Investigate how well complicated spring elements with 6 DOF perform, also when subjected to large rotations.
- Investigate how well brick elements perform.
- Investigate how a user subroutine can be constructed in OPENRADIOSS.
- Investigate why the current compilation flags do not improve the performance of the MPP version of OPENRADIOSS. It should be possible to optimise OPENRADIOSS more and obtain a performance improvement of a factor of two.
- A set of benchmark problems should be created for 3DCAV that can be used to investigate how well LS-DYNA and OPENRADIOSS are reproducing the correct solution.
- Couple 3DCAV with OPENRADIOSS to perform UNDEX simulations with OPENRADIOSS.

To run and allow the conversion of LS-DYNA models to OPENRADIOSS models, it is recommended to make an internal converter for TNO.

Given the large potential of using OPENRADIOSS in TNO-NOS, we recommend to starting a local collaboration network on OPENRADIOSS inside the Netherlands. This OPENRADIOSS-NL collaboration can be tremendously beneficial for the expansion of physics inside OPENRADIOSS and this can lead to PhD projects in collaboration within this OPENRADIOSS-NL collaboration.

7.3 Recommendations for Altair

Working in collaboration with Altair has been pleasant and productive. Besides the following recommendations we would like to stress that Altair has put a lot of effort in making OPENRADIOSS open source and that we can see that this is clearly helping the community a lot. Despite this, there are always points that can be improved and

we hope these can be included in the code. The recommendations for Altair are the following:

- Currently, OPENRADIOSS uses CI/CD to test if OPENRADIOSS compiles in 3 different situations for both the starter and engine in the case the codes are compiled with single precision, double precision or with openMPI. The developers of OPENRADIOSS started implementing another test named QA. It is unclear what this is but, the recommendation for OPENRADIOSS is to extend the CI/CD interface in a few different aspects:
  - Extend the compilation testing of the engine to including testing that the code compiles using the INTEL compiler.
  - Extend the CI/CD to include tests of simple test problems like the simulation of shell element tests with different shell elements such that the solution remains consistent. Similar thing can be done for testing beam elements and different spring elements.
  - Extend the CI/CD to include big test cases that are run for a specified number of time steps, for example 100- or 1000-time steps. Using this the code is not allowed to crash or cause a segmentation fault.
  - Extend the CI/CD to include unit tests for commonly used functions.

  Including these suggestions in OPENRADIOSS is expected to increase the trustworthiness of the code, make the code development easier, and will help prevent the accidental creation of bugs that could have been prevented.
- Because current variable names are often obscure, it is recommended to use the snake case naming convention for all variables and file names. Additionally, we also recommend that code that is implemented is properly documented and is not accepted if it is not well documented. This will make the code more readable and accessable for new users.
- The current file format of the time file outputs is in a .CSV file (comma-separated values). This is far from ideal because it is hard to read .CSV files because they need to be read in completely. This is because .CSV files are not structured. Instead it is recommended for Altair to implement the time file outputs in the HDF5 file format (HDF group, 2024). The main advantage of HDF5 is that files become structured, and it is not necessary to read in the complete file but only the data that is required. Secondly, it is straightforward to compress HDF5 files and therefore reduce the required storage space for HDF5 files. Fig. 7.1 shows the schematic structure of a possible future HDF5 format for the OPENRADIOSS time files. The HDF5 file is divided in different groups for the different element formulations and the nodes. For the different element formulations variables like stress and strain can be logged. The nodes can log variables like displacement, position, velocity, acceleration, and others. Each variable has two main attributes (atr.); firstly, the unit exponents for mass, time, and length such that in an automatic way the unit of each variable can be determined. Secondly, a short description for each variable is present such that users of the output understand what the variable represent. Similarly, there is a group called time which stores the time and its corresponding unit. An option is to make a link between the /TIME group to each variable group, such that the time is also a subgroup in each group. Storing the time centrally, also makes it possible to use different time stepping sizes for logging the data of nodes and shell for example. Two other important groups are the units, which stores the conversion between the internal units and the SI unit system. This will allow to read the HDF5 file and at the same time assign units to the variables (e.g. similar to Borrow and Borrisov, 2020). Secondly, a group called /INFO will be used to store information about

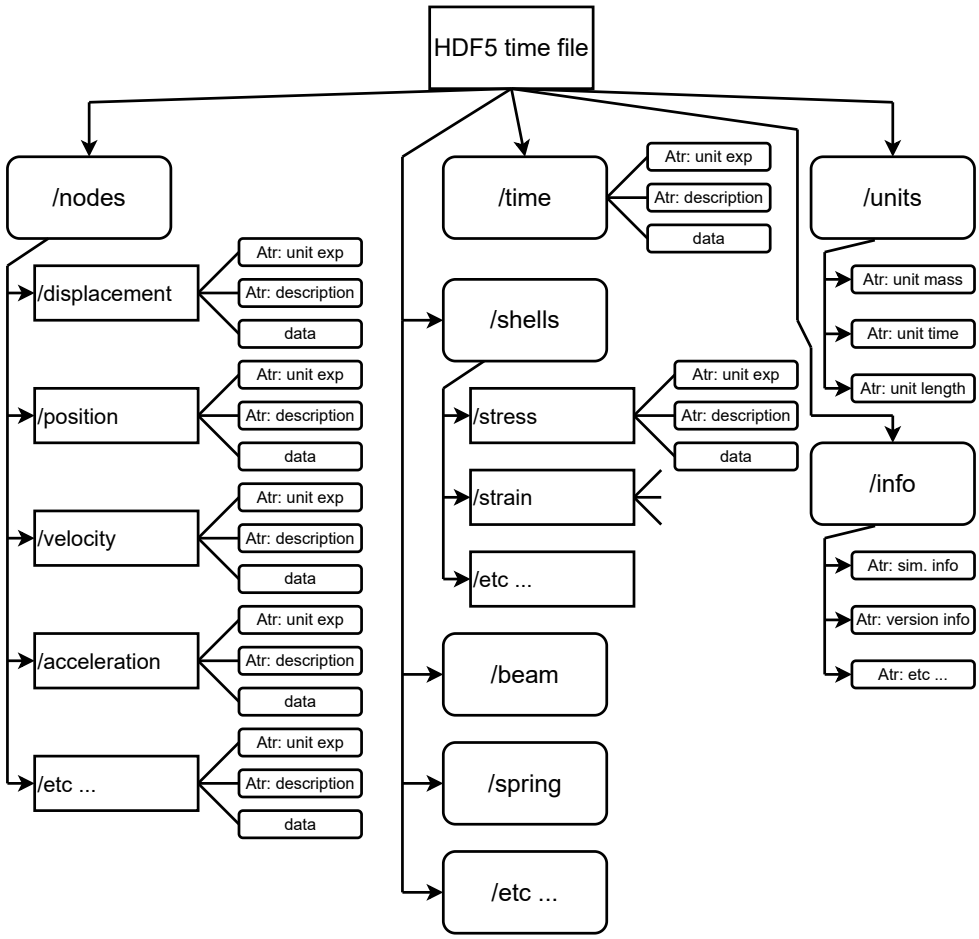


Figure 7.1 Schematic view of envisioned future time file structure for OpenRadioss using HDF5 files. Each HDF5 file has groups for the different element types and the nodes, which correspondingly have different subgroups with the different variables of that element type that are logged. These variables each have at least two attributes (Atr.), its unit exponents for mass, time, and length. Secondly, a short description of what the variable is. Lastly, the data is stored in the group. A separate group is present for the time. Also, a group is present for the units that are used based on the SI system and lastly an extra group is present for information about the simulation like the simulation itself, version of the software or other details.

the simulation itself and the version of OpenRadioss that was used.
More extensions for this format can be added like a separate group for energies, like the internal energy, contact energy, etc. such that energies can be easily accessed. Also adding groups for rigid bodies is an option.

- It is recommended to include the documentation of Radioss in OpenRadioss and require merge request to also update the different sections/chapters of the documentation of OpenRadioss. This will make the documentation for the users of OpenRadioss more up to date and will improve the quality of the documentation for the users of Radioss because the manual is updated at the same time that the merge request is done. This might require an initial big effort because if the manual is not written in a text format like LaTeX it might require initially a lot of work. However, in the long run having the manual in the Radioss repository will significantly strengthen the code.
- OpenRadioss should release different versions of the code that are tracking recent changes. The use of daily releases is not considered useful for the developers because they can access the daily version using GitHub. Also, it is

not useful that every day you download a different version of the code that is tagged daily. This makes the releases sensitive for the introduction of bugs, and it therefore should be avoided to daily release a version. It is recommended that OPENRADIOSS only releases a tagged version every month[2] with the new features compared to the previous version.

- OPENRADIOSS lacks fundamental examples. These are examples that test the 'fundamental' properties of a single element formulation or material model. These benchmark problems or tests are ideal to include in the CI/CD of OPENRADIOSS. Examples of following problems are recommended to include in OPENRADIOSS:
  - The Irons and Razzaque (1972) patch test (shell elements and solid elements).
  - Cantilever beam (like in this report) for both beam, shell and brick elements.
  - Cantilever beam with odd sized elements (e.g. fig. 4 of Macneal and Harder, 1985) with trapezoidal/parallelogram shaped elements for shell and brick elements.
  - The curved beam problem (e.g. fig. 5 of Macneal and Harder, 1985) for beam, shell and brick elements.
  - The twisted beam problem (e.g. fig. 6 of Macneal and Harder, 1985) for (beam), shell and brick elements. This problem might already be in the testing but is called 'smoke test'.
  - The Scordelis–Lo Roof (e.g. fig. 8 of Macneal and Harder, 1985) for shell elements.
  - The spherical shell problem (e.g. fig. 9 of Macneal and Harder, 1985) for shell elements.
  - The thick-walled cylinder (e.g. fig. 10 of Macneal and Harder, 1985) for brick elements.
  - The tensile test (example on OPENRADIOSS website for a single material model) for as many material models as possible.
  - Spring-mass system tests for linear and non-linear springs (like in this report).
  - Standard CFD test cases like Sedov blast, Sod (1978) shock tube, and the Noh (1987) problem.
  - A standard test for trusses.
  - Standard tests for different contact types.
- Real case scenarios of OPENRADIOSS are sometimes hard to find, therefore it is recommended that the publications that do use OPENRADIOSS are listed on the website (We are happy that OPENRADIOSS has done this a few months ago). This way people can see that OPENRADIOSS is widely used in industry and it is easy to see the more recent developments of OPENRADIOSS as a code. Additionally, it is recommended to include explicitly a directory of examples in the source code, similar to codes like SWIFTSIM (Schaller et al., 2023).
- OPENRADIOSS lacks the option to automatically create checkpoint files on demand. Having the option of automatically check pointing OPENRADIOSS will allow to do bigger simulations on HPC facilities that do not allow the code to be run for unlimited time. Furthermore, having automatic checkpoint files will make the code easier to debug.

[2] Or longer if no significant changes were made.

8 References

Abramowitz, M. and Stegun, I. (1965). *Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables*. Applied mathematics series. Dover Publications.

Altair (2022a). Altair radioss 2022: Reference manual.

Altair (2022b). Altair radioss 2022: Theory manual.

Altair (2022c). Industry-proven altair radioss finite element analysis solver now available as open-source solution.

Altair (2023). Altair radioss 2023: user guide.

Batoz, J.-L. and Dhatt, G. (1990). *Modélisation des structures par éléments finis*. Presses Université Laval.

Belytschko, T. and Leviathan, I. (1994). Physical stabilization of the 4-node shell element with one point quadrature. *Computer Methods in Applied Mechanics and Engineering*, 113(3-4):321 – 350.

Belytschko, T., Lin, J. I., and Chen-Shyh, T. (1984). Explicit algorithms for the nonlinear dynamics of shells. *Computer Methods in Applied Mechanics and Engineering*, 42(2):225 – 251. Cited by: 526.

Belytschko, T. and Tsay, C.-S. (1983). A stabilization procedure for the quadrilateral plate element with one-point quadrature. *International Journal for Numerical Methods in Engineering*, 19(3):405–419.

Benson, D. J. (2007). The history of ls-dyna.

Bini Leite, R., Barriga, A. d., Olivares, G., and Gomez, L. (2021). Scalability study of a large ditching model using radioss multi-domain technique to reduce computational time. In *AIAA AVIATION 2021 FORUM*, page 2993.

Birdsall, C. K. and Langdon, A. B. (1985). *Plasma physics via computer simulation*. CRC press.

Borrow, J. and Borrisov, A. (2020). swiftsimio: A python library for reading swift data. *Journal of Open Source Software*, 5(52):2430.

Branch, M. A., Coleman, T. F., and Li, Y. (1999). A subspace, interior, and conjugate gradient method for large-scale bound-constrained minimization problems. *SIAM Journal on Scientific Computing*, 21(1):1–23.

Brandão, G. (2023). Falling object protective structure (fops) using altair radioss.

Bulla, M., Kolling, S., and Sahraei, E. (2021). A material model for the orthotropic and viscous behavior of separators in lithium-ion batteries under high mechanical loads. *Energies*, 14(15):4585.

Bulla, M., Schmandt, C., Kolling, S., Kisters, T., and Sahraei, E. (2023). An experimental and numerical study on charged 21700 lithium-ion battery cells under dynamic and high mechanical loads. *Energies*, 16(1):211.

Cina, S. (2019). *Study of a seat in composite material: Fem modelling and solver influence*. PhD thesis, Politecnico di Torino.

Cole, R. (1948). *Underwater Explosions*. Dover books on engineering and engineering physics. Dover Publications.

Colfax Research (2017). A performance-based comparison of c/c++ compilers. *Colfax Research*.

Courant, R., Friedrichs, K., and Lewy, H. (1928). Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen*, 100:32–74.

Di Pasquale, E. (2015). On automatic crash model translation.

Ferry, E., Robert, A., and Loverini, M. (2023). 5.56x45 ss109 impact on an armox 500t plate.

Flanagan, D. and Belytschko, T. (1981). A uniform strain hexahedron and quadrilateral with orthogonal hourglass control. *International journal for numerical methods in engineering*, 17(5):679–706.

Franke, F., Slowik, T., Burger, U., and Hühne, C. (2022). Numerical investigation of drone strikes with various aircraft targets. In *AIAA SCITECH 2022 Forum*, page 2603.

Gauss, C. F. (1815). *Methodus nova integralium valores per approximationem inveniendi*. H. Dieterich (Gottingae).

Green, R. (2022). The next chapter for the intel fortran compiler. https://community.intel.com/t5/Blogs/Tech-Innovation/Tools/The-Next-Chapter-for-the-Intel-Fortran-Compiler/post/1439297.

Green, R. (2023). Ifx vs ifort performance difference. https://community.intel.com/t5/Intel-Fortran-Compiler/IFX-vs-IFORT-performance-difference/m-p/1471952.

Halbiniak, K., Wyrzykowski, R., Szustak, L., Kulawik, A., Meyer, N., and Gepner, P. (2022). Performance exploration of various c/c++ compilers for amd epyc processors in numerical modeling of solidification. *Advances in Engineering Software*, 166:103078.

Haufe, A., Schweizerhof, K., and DuBois, P. (2013). Properties & limits: Review of shell element formulations.

Haug, E. (1981). Engineering safety analysis via destructive numerical experiments. *Engineering Transactions*, 29(1).

Haug, E., Scharnhorst, T., and Du Bois, P. (1986). Fem-crash, berechnung eines fahrzeugfrontalaufpralls. *VDI Berichte*, 613:479–505.

HDF group (2024). The hdf5 library & file format.

Hooke, R. (1678). *Lectures de Potentia Restitutiva, Or of Spring Explaining the Power of Springing Bodies*. [Cutlerian lecture. John Martyn.

IFX developers (2023). Porting guide for intel fortran compiler. https://www.intel.com/content/www/us/en/developer/articles/guide/porting-guide-for-ifort-to-ifx.html.

Intel developers (2010). Quick-reference guide to optimization with intel compilers version 12.

Intel developers (2023). Intel oneapi dpc++/c++ compiler developer guide and reference.

Irons, B. M. and Razzaque, A. (1972). Experience with the patch test for convergence of finite elements. In Aziz, A., editor, *The Mathematical Foundations of the Finite Element Method with Applications to Partial Differential Equations*, pages 557–587. Academic Press.

IT Boston University (2023). Intel compiler flags.

Jezdik, R., Rulc, V., Kubovy, P., Kemka, V., Kovanda, J., Mlejnkova, B., and Tikkanen, T. (2023). Analysis of pedestrian-tram collision phenomena: Possibilities of simulation models validation. *MM Science Journal*, 2023-June:6594 – 6601. Cited by: 0; All Open Access, Bronze Open Access.

Jones, H. and Miller, A. R. (1948). The detonation of solid explosives: the equilibrium conditions in the detonation wave-front and the adiabatic expansion of the products of detonation. *Proc. R. Soc. Lond.*, (A194):480–507.

Kosloff, D. and Frazier, G. A. (1978). Treatment of hourglass patterns in low order finite element codes. *International Journal for Numerical and Analytical Methods in Geomechanics*, 2(1):57 – 72.

Le Métayer, O. and Saurel, R. (2016). The noble-abel stiffened-gas equation of state. *Physics of Fluids*, 28(4). Cited by: 97; All Open Access, Green Open Access.

Lee, E. L., Hornig, H. C., and Kury, J. W. (1968). Adiabatic expansion of high explosive detonation products.

Lequiniou, E. (2023). Personal communication.

Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168.

Love, A. E. H. (1888). The Small Free Vibrations and Deformation of a Thin Elastic Shell. *Philosophical Transactions of the Royal Society of London Series A*, 179:491–546.

Loverini, M. (2023a). Altair solution for ballistic impact on water tank.

Loverini, M. (2023b). Altair solutions for sphere hydroforming with explosive.

Loverini, M. (2023c). Altair solutions for vehicle assessment according to stanag regulation.

Loverini, M. (2023d). Multiple explosives detonation simulation.

Loverini, M. (2023e). Underwater explosion – validation study.

Loverini, M. and Robert, A. (2022). Air burst explosion.

Loverini, M. and Robert, A. (2023). Air burst explosion.

Macneal, R. H. and Harder, R. L. (1985). A proposed standard set of problems to test finite element accuracy. *Finite Elements in Analysis and Design*, 1(1):3–20.

Marquardt, D. W. (1963). An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441.

Meirovitch, L. and Wesley, D. A. (1967). On the dynamic characteristics of a variable-mass slender elastic body under high accelerations. *AIAA Journal*, 5(8):1439 – 1447. All Open Access, Green Open Access.

Mestres, E. (2023a). Radioss sample for welding line process.

Mestres, E. (2023b). threaded bolt process sample.

Mindlin, R. (1951). Influence of rotatory inertia and shear on flexural motions of isotropic, elastic plates.

Mittal, A., Mahato, A. C., Sharma, M., Mukhopadhyay, A., and Kadian, A. K. (2023). Explicit dynamic frontal crash analysis of an all-terrain vehicle roll cage. *Applications of Modelling and Simulation*, 7:85–92.

Nakano, R. (2023). Modeling foam material with /mat/law70.

Newton, I. (1687). *Philosophiae naturalis principia mathematica*. Early English books online. Jussu Societas Regiæ ac typis Josephi Streater, prostant venales apud Sam. Smith.

Noh, W. F. (1987). Errors for calculations of strong shocks using an artificial viscosity and an artificial heat flux. *Journal of Computational Physics*, 72(1):78–120.

Pasligh, N., Schilling, R., and Bulla, M. (2017). Modeling of rivets using a cohesive approach for crash simulation of vehicles in radioss. *SAE International journal of transportation safety*, 5(2).

Potter, D., Stadel, J., and Teyssier, R. (2017). PKDGRAV3: beyond trillion particle cosmological simulations for the next era of galaxy surveys. *Computational Astrophysics and Cosmology*, 4(1):2.

Prashanth, A. R. (2022). Rops test scenario analysis with radioss explicit.

Reddy, J. (2006). *Theory and Analysis of Elastic Plates and Shells*. CRC Press.

Reddy, J. (2017). Energy principles and variational methods in applied mechanics.

Reissner, E. (1945). The effect of transverse shear deformation on the bending of elastic plates.

Robert, A. and Loverini, M. (2023). Boat section slamming simulation.

Robert, A., Loverini, M., and Nordgren, F. (2023). Bottle drop simulation.

Sait, U. (2021). Ifort vs ifx speed issue. https://community.intel.com/t5/Intel-Fortran-Compiler/Ifort-vs-ifx-speed-issue/m-p/1295644.

Schaller, M., Borrow, J., Draper, P. W., Ivkovic, M., McAlpine, S., Vandenbroucke, B., Bahé, Y., Chaikin, E., Chalk, A. B. G., Chan, T. K., Correa, C., van Daalen, M., Elbers, W., Gonnet, P., Hausammann, L., Helly, J., Huško, F., Kegerreis, J. A., Nobels, F. S. J., Ploeckinger, S., Revaz, Y., Roper, W. J., Ruiz-Bonilla, S., Sandnes, T. D., Uyttenhove, Y., Willis, J. S., and Xiang, Z. (2023). Swift: A modern highly-parallel gravity and smoothed particle hydrodynamics solver for astrophysical and cosmological applications. *arXiv e-prints*, page arXiv:2305.13380.

Schaller, M., Gonnet, P., Chalk, A. B. G., and Draper, P. W. (2016). SWIFT: Using Task-Based Parallelism, Fully Asynchronous Communication, and Graph Partition-Based Domain Decomposition for Strong Scaling on more than 100,000 Cores. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, page 2.

Shamchi, S., Farahani, B. V., Bulla, M., and Kolling, S. (2024). Mechanical behavior of lithium-ion battery separators under uniaxial and biaxial loading conditions. *Polymers*, 16(8):1174.

Sharp, P. (2023a). Bird strike on windshield.

Sharp, P. (2023b). Bumper beam.

Sharp, P. (2023c). Openradioss user documentation.

Sharp, P. (2023d). Yaris impact on pole at 40km/h.

Sod, G. A. (1978). Review. A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws. *Journal of Computational Physics*, 27(1):1–31.

Solanki, K., Oglesby, D., Burton, C., Fang, H., and Horstemeyer, M. (2004). Crashworthiness simulations comparing pam-crash and ls-dyna. *SAE Technical Papers*. Cited by: 7.

Springel, V. (2005). The cosmological simulation code GADGET-2. MNRAS, 364(4):1105–1134.

Timoshenko, S. P. (1921). Lxvi. on the correction for shear of the differential equation for transverse vibrations of prismatic bars. *Philosophical Magazine Series 1*, 41:744–746.

Timoshenko, S. P. (1922). X. on the transverse vibrations of bars of uniform cross-section. *Philosophical Magazine Series 1*, 43:125–131.

Tofrowaih, K., Sulaiman, S., Samad, M. A., Azlan, K., Ab Razak, M., Rahman, M. A., Lubis, A. S., and Joehary, A. A. (2021). Evaluation of suv roof crush analysis using alternative non-linear structural analysis solver. *Journal of the Society of Automotive Engineers Malaysia*, 5(2):176–184.

Tuitman, J. T. (2023). Program to create 3DCAV fluid mesh for existing structural models – V1418 WP8.7 (U).

Uflyand, Y. S. (1948). The propagation of waves in the transverse vibrations of bars and plates. *Akad. Nauk. SSSR, Prikl. Mat. Mech*, 12(287-300):8.

Verlet, L. (1967). Computer “Experiments” on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159(1):98–103.

Walter, J. W. and Bellshaw, D. (1993). Survey to determine the value of dyna.

Wienholtz, O., Villeneuve, S., and Altair, L. (2023). How to build openradioss.

Wilkins, M. L., Squier, B., and Halperin, B. (1964). The equation of state of pbx 9404 and lx04-01.

Zeng, Q. and Combescure, A. (1998). New one-point quadrature, general non-linear quadrilateral shell element with physical stabilization. *International Journal for Numerical Methods in Engineering*, 42(7):1307 – 1338. Cited by: 31.

Zheng, J., He, Y., Zhao, M., and Xia, J. (2023). Dynamic response analysis of spherical pressure hull implosion inside adjacent underwater structure. *Ocean Engineering*, 283. Cited by: 0.

Zupan, D. and Saje, M. (2004). On “a proposed standard set of problems to test finite element accuracy”: the twisted beam. *Finite Elements in Analysis and Design*, 40(11):1445–1451.

9 Approval

F.S.J. Nobels M.Sc.	(author)	-	ValidSigned by Folkert Nobels on 2025-05-15 12:41:04 .....
dr ir J.T. Tuitman	(checked)	-	ValidSigned by Johan Tuitman on 2025-05-15 14:39:02 .....
ir W. Trouwborst	(checked)	-	ValidSigned by Wim Trouwborst on 2025-05-15 20:26:41 .....
R.C. Dragt M.Sc.	(approved)	-	ValidSigned by Sander Dragt on 2025-05-15 14:15:31 .....

A Getting started manual

For the internal flow of working with OpenRadioss a starting manual was created. The starting manual describes the different aspects of the code that are commonly used and gives a beginning user a good perspective on how to work with the code. The main features in the manual that are discussed are:

- How to get the code.
- How to know the version that you use.
- What are the dependencies of the code.
- How to get all the dependencies on the TNO-NOS machines.
- What are additional configuration options.
- How to speed up the code.
- How to run the code.
- How to run an example.
- What are the different runtime options.
- How to convert the output.
- How to analyse the output.

Getting started OpenRadioss

Folkert Nobels (TNO Naval and Offshore structures)

January 29, 2024

Getting The Code

OpenRadioss contains large files that use git lfs. Git lfs (Git Large File Storage) needs to be initialized as:

```
git lfs install
```

The code is available from the GitHub repository. You can download OpenRadioss by downloading it from the following location:

```
git clone https://github.com/OpenRadioss/OpenRadioss.
    git
```

Keeping track of the version

The version of OpenRadioss changes rapidly, because of this you should keep track of the current version that you use. You can find the current version by

```
git log
```

The first line of this shows the current version that you use which is the long combination of letters and numbers after commit.

Getting Help

Currently Folkert Nobels knowns the most about OpenRadioss, so questions on how to start with OpenRadioss can be asked to him. Specific questions can be asked online by creating an issue on GitHub.

Dependencies

Git-LFS

Git Large File Storage is used to download the large files in the repository.

cmake and make

Cmake version 3.20.4 and make version 4.2.1 or higher.

Perl and Python

Perl version 5.26 or higher and python 3.

Gfortran

Version 11.x or higher of gfortran is required. Gfortran is used to compile the code.

IFort or IFx

For the fortran intel compilers version 2021 or higher is required (and tested).

g++, cpp and gcc

Version 11.x or higher is required.

Using version 12 of gfortran, g++, cpp and gcc

On Goldfish (openSUSE) you can use version 12 of these software packages by including the following softlinks in your `/bin/` directory

```
ln -s  /usr/bin/gfortran-12 gfortran
ln -s /usr/bin/cpp-12 cpp
ln -s /usr/bin/g++-12 g++
ln -s /usr/bin/gcc-12 gcc
```

Using the intel compiler on Goldfish and Marlin

On Goldfish add:

```
alias intel='source /uhome/tuitmanjt/intel/oneapi/
    setvars.sh intel64'
```

And on Marlin add:

```
alias intel='source /Applications/oneapi2023/setvars.
    sh intel64'
```

To your rc file, and initialize this by running intel in the terminal.

Initial Setup (SMP)

Here we describe the basics of setting the code up to be run on a single node with shared-memory multiprocessing (SMP). OpenRadioss uses `git-lfs` and Cmake for setup. To get a basic version of the starter on most platforms, run

```
cd starter
./build_script.sh -arch=linux64_gf
```

The exectuble of the starter can be found in

```
OpenRadioss/exec
```

To get a basic version of the engine on most platforms, run cd engine

```
./build_script.sh -arch=linux64_gf
```

Or

```
./build_script.sh -arch=linux64_intel
```

Useful Engine Configuration Options

A description of the available options can be found by using `./build_script.sh`.

```
-prec=[dp|sp]
```

There is the option to compile OpenRadioss in single and double precision. Double precision is the default, single precision is 40% faster.

```
-mpi=[ompi|impi]
```

The option to use OpenMPI or MPI to get communication between different computing nodes or different NUMA regions (e.g. a single chip).

```
-debug=[0|1]
```

Debug version of the code, 0 is no debug flags (default) and 1 has the usual debug flags.

```
-verbose
```

Increase the verbosity of the output during the build. Speeding up the code with compiler flags It is possible to speed-up OpenRadioss significantly when using the intel compiler and when using different compiler flags on lines 77 and 78 of `./engine/CMake_Compilers/cmake_linux64_intel.txt`. Replace `-axSSE3,COMMON-AVX512 -no-fma -O3` with `-no-fma -Ofast -xHost -static`

Running the Code (SMP)

After compilation, you will be left with two binaries. One is the called `starter_linux_gf` and the other is either

called `engine_linux64_gf` or `engine_linux64_intel`. The starter is used to create the simulation that will be simulated and the engine is used to do the actual simulation. Before running simulations you need to initialize a few variables in your `.bashrc` file:

```
export OPENRADIOSS_PATH=/path/to/OpenRadioss
export RAD_CFG_PATH=$OPENRADIOSS_PATH/hm_cfg_files
export LD_LIBRARY_PATH=$OPENRADIOSS_PATH/extlib/
    hm_reader/linux64/:$OPENRADIOSS_PATH/extlib/h3d/
    lib/linux64/:$LD_LIBRARY_PATH
```

Running an Example

The OpenRadioss example repository contains a large number of examples that you can run. For example the cantilever beam simulation. Each simulation is run in two steps, the first step is running the starter

```
./dir_to_exec/starter_linux64_gf -i filename_0000.rad
```

This is followed by using the engine to run the simulation as

```
./dir_to_exec/engine_linux64_intel -i filename_0001.
    rad
```

Or

```
./dir_to_exec/engine_linux64_gf -i filename_0001.rad
```

Initial Setup (MPI)

Especially when using big models, simulations will require using more than one numa region. This means that the user wants to use a version of the code that uses Message Passing Interface (MPI). Obtaining the software on Goldfish and Marlin is identical to the description above. The main difference is that compiling the code is different. For our servers we only use the intel MPI version of OpenRadioss. The engine can be build as:

```
./build_script.sh -arch=linux64_intel -mpi=impi
```

Once, this is completed, your MPI version of OpenRadioss is build.

Running the Code (MPI)

Running the code is different, because you will need to launch several copies of the same program at the same time (i.e. in jargon, several MPI ranks). Firstly, you need to specify the number of desired threads:

```
export OMP_NUM_THREADS=<Nthreads>
```

followed by running the starter for the amount of MPI ranks that you want:

```
./dir_to_exec/starter_linux64_gf -np <N_MPI> -i
    filename_0000.rad
```

where `<N_MPI>` indicates the number of MPI ranks. This is followed by running the code over mpi using the right number of MPI ranks as:

```
mpirun -np <N_MPI> --map-by socket:PE=<Nthread> --bind
    -to core /dir_to_exec/engine_linux64_intel_impi -i
     filename_0001.rad
```

Runtime options

All the run time options can be shown by `./dir_to_exec/engine_linux64_gf -help`. Some useful options are:

```
-nthread [integer] / -nt [integer]
```

This sets the number of SMP threads per Single Program Multiple Data (SPMD) domain.

```
-version / -v
```

Get the version of the current engine, note that this is not a replacement for `git log`.

```
-output=[PATH]
```

Set the output file directory for all output and created files. Controlling the time step It is possible to control the minimum time step and the Courant-Friedrichs-Lewy (CFL) condition by:

```
/DT/NODA/CST/0
CFL_constant      dTmin
```

Where the CFL constant by default is 0.9, but we find that you should use at most 0.75. dTmin specifies the minimum allowed timestep. Converting the output OpenRadioss has three types of outputs. The first type is `*.h3d` files. In general, we do not use this file type. Instead we use the `ANIM` files or `*A*` files. Both types contain the same information, only the `ANIM` files can be converted to `*.vtk` files, which can be opened with `paraview`. During runtime the user needs to specify which information they want to include for each element. In order, to get the `ANIM` to `vtk` converter, go to `tools/anim_to_vtk/linux64` and execute

```
./build.bash
```

This creates the `ANIM` converter in your exec directory named `anim_to_vtk_linux64_gf`. Now you can convert `ANIM` files as

```
./dir/exec/anim_to_vtk_linux64_gf filenameAnumber >
    filename_number.vtk
```

You can optionally add an alias in your `.bashrc` file (or other rc file e.g. `.zshrc`) as:

```
alias anim_to_vtk=/dir/exec/anim_to_vtk_linux64_gf
```

Such that you can convert files simply as

```
anim_to_vtk filenameAnumber > filename_number.vtk
```

The third type of output are time files, `TH` files or `*T*` files. These files store a smaller number of elements at a much higher frequency than the `ANIM` and `*.h3d` files. The time files are therefore ideal if you want to know in detail what is happening to a large number of elements. The time files can be converted into a `*.csv` file. To get the `TH` file to `*.csv` converter go to `tools/th_to_csv/linux64` and execute

```
./build.bash
```

This creates the `th_to_csv` converter in your exec directory named `th_to_csv_linux64_gf`. Converting `TH` files can be done as

```
./dir/exec/th_to_csv_linux64_gf filenameT01
```

You can optionally add an alias in your `.bashrc` file:

```
alias th_to_csv=/dir/exec/th_to_csv_linux64_gf
```

The `ANIM` converter can be written such that it converts all files by adding the following code to your `.bashrc`:

```
anim_to_vtk='/dir/to/OpenRadioss/exec/
    anim_to_vtk_linux64_gf'
anim_to_vtk_all () {
  number_of_files=$(ls -1R $1A* | wc -l)
  maximum_index=$((number_of_files))
  start_index=1
  for (( i=$start_index; i<=$maximum_index; i++ ))
  do
    printf -v output_i "%03d" $i
    $anim_to_vtk $1A$output_i > $1_$output_i.vtk
  done
}
```

This way you can run to convert all `ANIM` files

```
anim_to_vtk_all filename
```

This can be combined in a single function such that also the time file is converted:

```
alias th_to_csv= /dir/to/OpenRadioss/exec/
    th_to_csv_linux64_gf
convert_radioss () {
  anim_to_vtk_all $1
  th_to_csv $1T01
}
```

Such that all files can be converted as

```
convert_radioss filename
```

Running and converting in one line If you want to run radioss (starter+engine) and the converter at the same time you can create a function:

```
radioss () {
  OR_starter -i $1_0000.rad
  OR_engine -i $1_0001.rad -nt $2
  anim_to_vtk_all $1
  th_to_csv $1T01
}
```

Now you can run radios on 4 cores as:

```
radioss filename 4
```

Analysing the `VTK` files The `VTK` files can be analysed using Paraview. Paraview is installed on all our Linux servers and you can send a software request for Paraview on windows. For more information see www.paraview.org/tutorials/ The properties of the `VTK` files are specified in the engine file, i.e. `filename_0001.rad`. The most important setting is the start time and time interval which can be specified by

```
/ANIM/DT
T_start  T_freq
```

Variables that you want to output can be specified by:

```
/ANIM/NODA/<NAME>
/ANIM/VEC/<NAME>
```

For nodal scalar data, vector data for your specified variable. Other options are available in the manual.

Analysing the `CSV` files

The time files can be extremely long, because of this TNO wrote their own internal python module to analyse these files. The package is named openradiosstimefilereader. This package reads in the *.csv file and creates a class such that in this class you can call each individual node or spring as:

```
timefile = OpenRadiossTimeFile("filenameT01.csv",
    filename_0000.rad)
quantity = timefile.nodes["id"]["quantity"]
```

or

```
quantity = timefile.springs["id"][quantity]
```

This allows the user to only know the id number of the element and not needing to look up the exactly matching column/row. Variables that are the same for each node like time can be simply called by `timefile.time()`. Warning: It is possible to load `CSV` files into excel, only when the file is too large it does only load part of the file. If your time file has more than 1000 lines this is already a problem. Because of this we recommend to use the module instead of using excel.

Example overview

The Radioss examples repository contains a large number of simple test cases that can be easily downloaded and used.

Advanced options

Control file

The control file is an optional file that can be created while a run is ongoing to safely stop the simulation or request additional outputs. Control files are named filename.cfl. In general, the control file is executed as quickly as possible. There are two optional keywords that allow it to stop only at a specific cycle or time:

```
/CYCLE/cyclenumber
/TIME/timevalue
```

In order to kill the job you can use:

```
/STOP
```

This will kill the job and creates a restart file. It is also possible to create more outputs, for that see the manual.

B OpenRadioss time file reader

OpenRadioss has the option to output more information of nodes, beams, shells or other elements. This is a similar feature to the DEFINENAME feature of LS-DYNA which is extensively used in 3DCAV simulations. Currently, the files that are created can have many columns and because of this selecting the right columns is prone to mistakes. Because of this a simple PYTHON library was created which reads the desired columns based on the file itself instead of needing to look up the correct column yourself. At the same time, the package also reads the correct units of the different variables. This prevents that mistakes are made with units.

```python
#!/usr/bin/env python3
import numpy as np
from unyt import kg, mm, ms, s, m, cm, unyt_quantity
import unyt
import csv
import re
import pandas as pd
import time

class OpenRadiossTimeFile:

  def __init__(self, csv_file_name, rad_file_name):
    # Store the string names of the files in the object
    self.csv_file_name = csv_file_name
    self.rad_file_name = rad_file_name

    # Load the internal units of the file
    self.load_internal_units()

    # open the file and store the data in variable data (use pandas because it is x10 faster)
    self.data = pd.read_csv(self.csv_file_name) #, engine="pyarrow"
    self.data = self.data.to_numpy()

    with open(csv_file_name, 'r') as f:
      for i, row in enumerate(csv.reader(f)):
        self.csv_column_names = row
        if i==0:
          break

    self.possible_numbers = []
    for i, column_name in enumerate(self.csv_column_names):
      new_element = re.findall(f"\d+", column_name)
      if not new_element:
        continue
      else:
        self.possible_numbers.append(new_element[0])

    # Array of the possible node numbers
    self.possible_numbers = list(set(self.possible_numbers))

    # Set all types to False
    self.file_has_nodes = False
    self.file_has_spring = False
    self.file_has_shell = False
    self.file_has_beam = False

    # Determine if we have nodes, springs, beams, shells, etc.
    for i, column_name in enumerate(self.csv_column_names):
      if "Nodes" in column_name:
        self.file_has_nodes = True
      elif "spring" in column_name:
        self.file_has_spring = True
      elif "shell" in column_name:
        self.file_has_shell = True
      elif "beam" in column_name:
        self.file_has_beam = True
      else:
        continue # unsupported element type or global variable


    # Set an empty dictionary and add keys of possible numbers to it
    if self.file_has_nodes:
      self.nodes = {}
      for key in self.possible_numbers:
        self.nodes[key] = {}

    if self.file_has_spring:
      self.springs = {}
      for key in self.possible_numbers:
        self.springs[key] = {}

    for j, used_number in enumerate(self.possible_numbers):
      #print(used_number)
      for i,column_name in enumerate(self.csv_column_names):
        type_name = column_name[-20:-1]
        type_name = type_name.strip()
        if (" "+used_number+" " in column_name) and ("Nodes" in column_name):

          # Determine which unit is used
          if type_name[0] == "F":
            unit = self.unit_F
          elif type_name[0] == "D" or type_name[0] == "X" or type_name[0] == "Y" or type_name[0] == "Z"
              :
            unit = self.unit_L
          elif type_name[0] == "V":
            unit = self.unit_V
```

```
            elif type_name[0] == "M":
              # Is this indeed the correct unit?
              unit = self.unit_Moment
            elif type_name[0] == "A":
              unit = self.unit_A
            elif type_name[0] == "R":
              unit = unyt.rad
            else:
              unit = 1.

            # set the information of this node in its dictionary
            self.nodes[used_number][type_name.lower()] = self.data[:,i] * unit
          elif (" "+used_number+" " in column_name) and ("spring" in column_name):

            # Determine which unit is used
            if type_name[0] == "0":
              unit = 1.
            elif type_name[0] == "F":
              unit = self.unit_F
            elif type_name[0] == "M":
              unit = self.unit_Moment
            elif type_name[0] == "L":
              unit = self.unit_L
            elif type_name[0] == "R":
              unit = unyt.rad
            elif type_name[0] == "I":
              unit = self.unit_E
            else:
              unit = 1.
            # set the information of this spring in its dictionary
            self.springs[used_number][type_name.lower()] = self.data[:,i] * unit



    def load_internal_units(self):

      # Open the file pointer
      fp = open(self.rad_file_name, "r")

      # Get all the lines of the file
      lines = fp.readlines()

      # loop over the file
      for i, line in enumerate(lines):
        if line.strip()=="/BEGIN":
          count = i

          string_units = lines[i+6].strip()
          string_last_two_units = string_units[5:].strip()
          string_last_unit = string_last_two_units[5:].strip()

          # unit of mass
          self.unit_M = unyt_quantity(1., string_units[:5].strip())
          # unit of length
          self.unit_L = unyt_quantity(1., string_last_two_units[:5].strip())
          # unit of time
          self.unit_T = unyt_quantity(1., string_last_unit[:5].strip())

          break

      # unit of velocity
      self.unit_V = self.unit_L / self.unit_T
      # unit of frequency
      self.unit_Freq = 1. / self.unit_T
      # unit of acceleration
      self.unit_A = self.unit_L / self.unit_T**2
      # unit of energy
      self.unit_E = self.unit_M * self.unit_L**2 / self.unit_T**2
      # unit of momentum
      self.unit_P = self.unit_M * self.unit_L / self.unit_T
      # unit of force
      self.unit_F = self.unit_M * self.unit_L / self.unit_T**2
      # unit of Inertia
      self.unit_I = self.unit_M * self.unit_L**2
      # unit of moment
      self.unit_Moment = self.unit_F * self.unit_L

    def time(self):
      return self.data[:,0] * self.unit_T

    def internal_energy(self):
      return self.data[:,1] * self.unit_E

    def kinetic_energy(self):
      return self.data[:,2] * self.unit_E

    def momentum_x(self):
      return self.data[:,3] * self.unit_P
    def momentum_y(self):
      return self.data[:,4] * self.unit_P
    def momentum_z(self):
      return self.data[:,5] * self.unit_P

    def momentum(self):
      return self.data[:,3:6] * self.unit_P

    def mass(self):
      return self.data[:,6] * self.unit_M

    def time_step(self):
      return self.data[:,7] * self.unit_T

    def rotation_energy(self):
      return self.data[:,8] * self.unit_E

    def external_work(self):
      return self.data[:,9] * self.unit_E

    def spring_energy(self):
      return self.data[:,10] * self.unit_E
```

```
    def contact_energy(self):
      return self.data[:,11] * self.unit_E

    def hourglass_energy(self):
      return self.data[:,12] * self.unit_E

    def initialise_displacement_variables(self, start_id, end_id, time_fraction=0.5):
      self.z_median = np.zeros(end_id - start_id + 1) * self.unit_L
      self.x_median = np.zeros(end_id - start_id + 1) * self.unit_L
      self.z_avg = np.zeros(end_id - start_id + 1) * self.unit_L
      self.x_avg = np.zeros(end_id - start_id + 1) * self.unit_L
      self.z_s_up = np.zeros(end_id - start_id + 1) * self.unit_L
      self.z_s_down = np.zeros(end_id - start_id + 1) * self.unit_L

      from_fraction = 1- time_fraction

      self.length_array = len(self.time())

      for index, node_id in enumerate(range(start_id, end_id + 1)):
        z_node = self.nodes[f"{node_id:d}"]["dz"]
        x_node = self.nodes[f"{node_id:d}"]["x"]

        self.z_avg[index] = np.average(z_node[int(self.length_array*from_fraction):])
        self.x_avg[index] = np.average(x_node[int(self.length_array*from_fraction):])
        self.z_median[index] = np.median(z_node[int(self.length_array*from_fraction):])
        self.x_median[index] = np.median(x_node[int(self.length_array*from_fraction):])

        self.z_s_down[index] = np.percentile(z_node[int(self.length_array*from_fraction):], 16)
        self.z_s_up[index] = np.percentile(z_node[int(self.length_array*from_fraction):], 84)

    def initialise_displacement_variables_y(self, start_id, end_id, time_fraction=0.5):
      self.y_median = np.zeros(end_id - start_id + 1) * self.unit_L
      self.x_median = np.zeros(end_id - start_id + 1) * self.unit_L
      self.y_avg = np.zeros(end_id - start_id + 1) * self.unit_L
      self.x_avg = np.zeros(end_id - start_id + 1) * self.unit_L
      self.y_s_up = np.zeros(end_id - start_id + 1) * self.unit_L
      self.y_s_down = np.zeros(end_id - start_id + 1) * self.unit_L

      from_fraction = 1- time_fraction

      self.length_array = len(self.time())

      for index, node_id in enumerate(range(start_id, end_id + 1)):
        y_node = self.nodes[f"{node_id:d}"]["dy"]
        x_node = self.nodes[f"{node_id:d}"]["x"]

        self.y_avg[index] = np.average(y_node[int(self.length_array*from_fraction):])
        self.x_avg[index] = np.average(x_node[int(self.length_array*from_fraction):])
        self.y_median[index] = np.median(y_node[int(self.length_array*from_fraction):])
        self.x_median[index] = np.median(x_node[int(self.length_array*from_fraction):])

        self.y_s_down[index] = np.percentile(y_node[int(self.length_array*from_fraction):], 16)
        self.y_s_up[index] = np.percentile(y_node[int(self.length_array*from_fraction):], 84)

    def set_FFT_spectrum(self, frequencies, abs_FFT):
      self.frequencies = frequencies
      self.absolute_FFT = abs_FFT

    # Function that can be used to fit the displacement
    def func_disp(t, A, t_decay, omega, phi):
      return A * np.sin(omega * t + phi) * np.exp(- t / t_decay)

if __name__ == "__main__":
  testdata = OpenRadiossTimeFile("tensile_LAW2_BIQUADT01.csv", "tensile_LAW2_0000.rad")


  print(testdata.time())

  print(testdata.n102_DX)
  print(testdata.n102_DY)
  print(testdata.n616_DX)
  print(testdata.n616_DY)
```

Additionally, for this project an additional file of routines was written to specifically compare different datasets. This also has routines that create specific plots that were made in this report.

```
#!/usr/bin/env python3
import numpy as np
import matplotlib.pyplot as plt
import unyt
import numpy.fft as fft
import matplotlib
import scipy.optimize as sco
import scipy.interpolate as sci
import os
import sys

matplotlib.use("Agg")

from load_csv_file import OpenRadiossTimeFile

# Plot parameters
params = {
    "axes.labelsize": 10,
    "axes.titlesize": 10,
    "font.size": 9,
    "font.family": "serif",
    "legend.fontsize": 9,
    "xtick.labelsize": 10,
    "ytick.labelsize": 10,
    "text.usetex": True,
    "figure.figsize": (5.15, 4.15),
    "figure.subplot.left": 0.15,
    "figure.subplot.right": 0.97,
    "figure.subplot.bottom": 0.13,
```

```
    "figure.subplot.top": 0.97,
    "figure.subplot.wspace": 0.15,
    "figure.subplot.hspace": 0.12,
    "lines.markersize": 6,
    "lines.linewidth": 1.0,
}
matplotlib.rcParams.update(params)

class DatasetCollection:
  def __init__(self, directory, base_name, label_name, index_to_consider, default_save_dir=None):
    # store the initialising variables in the class
    self.directory = directory
    self.base_name = base_name
    self.label_name = label_name
    self.index_to_consider = index_to_consider

    # set the default save directory to None if not specified
    self.save_dir = default_save_dir

    # make the colour array one more longer than the amount of directories
    # we do this because the last yellow colour is a bit difficult to see
    self.colour_array = plt.cm.plasma(np.linspace(0,1,len(self.directory)))

    # Load all the different datasets in this class using the dataset class
    self.dataset_array = []
    for direc, name in zip(self.directory, self.base_name):
      dataset = OpenRadiossTimeFile(f"{direc}/{name}T01.csv", f"{direc}/{name}_0000.rad")
      self.dataset_array.append(dataset)

  def plot_vertical_displacement(self):
    # Loop over the different datasets
    for dataset, node_id, label, colour in zip(self.dataset_array, self.index_to_consider, self.
        label_name, self.colour_array):
      # get the time
      time = dataset.time()

      # get the vertical displacement
      z = dataset.nodes[f"{node_id:d}"]["dz"]

      # Plot the resulting curve for the vertical displacement
      plt.plot(time, z, label=label, color=colour)

    return time.units, z.units

  def plot_internal_energy(self):
    # Loop over the different datasets
    for dataset, node_id, label, colour in zip(self.dataset_array, self.index_to_consider, self.
        label_name, self.colour_array):
      # get the time
      time = dataset.time()

      # get the vertical displacement
      internal_energy = dataset.internal_energy()

      # Plot the resulting curve for the vertical displacement
      plt.plot(time, internal_energy, label=label, color=colour)

    return time.units, internal_energy.units

  def plot_kinetic_energy(self):
    # Loop over the different datasets
    for dataset, node_id, label, colour in zip(self.dataset_array, self.index_to_consider, self.
        label_name, self.colour_array):
      # get the time
      time = dataset.time()

      # get the vertical displacement
      kinetic_energy = dataset.kinetic_energy()

      # Plot the resulting curve for the vertical displacement
      plt.plot(time, kinetic_energy, label=label, color=colour)

    return time.units, kinetic_energy.units

  def check_that_we_can_save(self, save_dir):
    # check that we have a place to store the figures
    if save_dir is not None:
      self.save_dir = save_dir
    elif self.save_dir is None:
      raise ValueError("The save directory should be defined when making plots!")

  def make_vertical_displacement_plot(self, y_mean=-0.78, x_max = 200., save_dir=None):

    self.check_that_we_can_save(save_dir)

    matplotlib.rcParams.update({"figure.figsize": (10.15, 4.15)})

    # Call the function to make a plot for each dataset
    time_unit, z_unit = self.plot_vertical_displacement()

    plt.axhline(y=y_mean, linestyle="--", color="k", label="Theoretical displacement")
    plt.legend()
    plt.xlim(0,x_max)
    plt.ylim(2*y_mean,-0.1*y_mean)
    plt.xlabel(f"time [${time_unit.latex_repr}$]")
    plt.ylabel(f"displacement [${z_unit.latex_repr}$]")
    plt.savefig(f"{self.save_dir}/cantileverbeamtest_100percent.pdf")
    plt.close()

    matplotlib.rcParams.update({"figure.figsize": (10.15, 4.15)})

    # Call the function to make a plot for each dataset
    time_unit, z_unit = self.plot_vertical_displacement()

    plt.axhline(y=y_mean, linestyle="--", color="k", label="Theoretical displacement")
    plt.legend()
    plt.xlim(0,x_max / 2.)
    plt.ylim(2*y_mean,-0.25*y_mean)
    plt.xlabel(f"time [${time_unit.latex_repr}$]")
    plt.ylabel(f"displacement [${z_unit.latex_repr}$]")
    plt.savefig(f"{self.save_dir}/cantileverbeamtest_50percent.pdf")
```

```
        plt.close()

        matplotlib.rcParams.update({"figure.figsize": (5.15, 4.15)})
        # Call the function to make a plot for each dataset
        time_unit, z_unit = self.plot_vertical_displacement()

        plt.axhline(y=y_mean, linestyle="--", color="k", label="Theoretical displacement")
        plt.legend()
        plt.xlabel(f"time [${time_unit.latex_repr}$]")
        plt.ylabel(f"displacement [${z_unit.latex_repr}$]")
        plt.xlim(0,x_max/10.)
        plt.ylim(2*y_mean,-0.25*y_mean)
        plt.savefig(f"{self.save_dir}/cantileverbeamtest_10percent.pdf")
        plt.close()

        matplotlib.rcParams.update({"figure.figsize": (5.15, 4.15)})

        # Call the function to make a plot for each dataset
        time_unit, z_unit = self.plot_vertical_displacement()

        plt.axhline(y=y_mean, linestyle="--", color="k", label="Theoretical displacement")
        plt.legend()
        plt.xlim(0,0.3 * x_max)
        plt.ylim(2*y_mean,0.0)
        plt.xlabel(f"time [${time_unit.latex_repr}$]")
        plt.ylabel(f"displacement [${z_unit.latex_repr}$]")
        plt.savefig(f"{self.save_dir}/cantileverbeamtest_30percent.pdf")
        plt.close()

    def make_internal_energy_plot(self, x_max = 200., save_dir=None, y_lim=None):

        self.check_that_we_can_save(save_dir)

        matplotlib.rcParams.update({"figure.figsize": (10.15, 4.15)})

        # Call the function to make a plot for each dataset
        time_unit, internal_energy_unit = self.plot_internal_energy()

        plt.legend()
        plt.xlim(0,x_max)
        if y_lim is not None:
          plt.ylim(y_lim[0], y_lim[1])
        plt.xlabel(f"time [${time_unit.latex_repr}$]")
        plt.ylabel(f"internal energy [${internal_energy_unit.latex_repr}$]")
        plt.savefig(f"{self.save_dir}/internal_energy_100percent.pdf")
        plt.close()

        matplotlib.rcParams.update({"figure.figsize": (10.15, 4.15)})

        # Call the function to make a plot for each dataset
        time_unit, internal_energy_unit = self.plot_internal_energy()

        plt.legend()
        plt.xlim(0,0.5*x_max)
        if y_lim is not None:
          plt.ylim(y_lim[0], y_lim[1])
        plt.xlabel(f"time [${time_unit.latex_repr}$]")
        plt.ylabel(f"internal energy [${internal_energy_unit.latex_repr}$]")
        plt.savefig(f"{self.save_dir}/internal_energy_50percent.pdf")
        plt.close()

    def make_kinetic_energy_plot(self, x_max = 200., save_dir=None, y_lim=None):

        self.check_that_we_can_save(save_dir)

        matplotlib.rcParams.update({"figure.figsize": (10.15, 4.15)})

        # Call the function to make a plot for each dataset
        time_unit, kinetic_energy_unit = self.plot_kinetic_energy()

        plt.legend()
        plt.xlim(0,x_max)
        if y_lim is not None:
          plt.ylim(y_lim[0], y_lim[1])
        plt.xlabel(f"time [${time_unit.latex_repr}$]")
        plt.ylabel(f"kinetic energy [${kinetic_energy_unit.latex_repr}$]")
        plt.savefig(f"{self.save_dir}/kinetic_energy_100percent.pdf")
        plt.close()

        matplotlib.rcParams.update({"figure.figsize": (10.15, 4.15)})

        # Call the function to make a plot for each dataset
        time_unit, kinetic_energy_unit = self.plot_kinetic_energy()

        plt.legend()
        plt.xlim(0,0.5*x_max)
        if y_lim is not None:
          plt.ylim(y_lim[0], y_lim[1])
        plt.xlabel(f"time [${time_unit.latex_repr}$]")
        plt.ylabel(f"kinetic energy [${kinetic_energy_unit.latex_repr}$]")
        plt.savefig(f"{self.save_dir}/kinetic_energy_50percent.pdf")
        plt.close()

    def set_problem_parameters(self, force, youngs_modulus, b, d, l, density, type_of_problem="cantilever
        "):
        # Pick the first dataset which we use for the unit system
        dataset = self.dataset_array[0]

        self.force = force * dataset.unit_F
        self.E = youngs_modulus * dataset.unit_F / dataset.unit_L**2
        self.b = b * dataset.unit_L
        self.d = d * dataset.unit_L
        self.l = l * dataset.unit_L
        self.area = self.b * self.d
        self.moment_of_inertia = self.b * self.d**3 / 12.
        self.density = density * dataset.unit_M / dataset.unit_L**3


        self.base_frequency = np.sqrt(self.E * self.moment_of_inertia / (self.density * self.area * self.l
            **4))

```

```
    self.natural_frequencies = np.zeros(5) / dataset.unit_T

    if type_of_problem=="cantilever" or type_of_problem=="clamped-free":
      self.natural_frequencies[0] = 3.52 * self.base_frequency
      self.natural_frequencies[1] = 22.4 * self.base_frequency
      self.natural_frequencies[2] = 61.7 * self.base_frequency
      self.natural_frequencies[3] = 121.0 * self.base_frequency
      self.natural_frequencies[4] = 200.0 * self.base_frequency
      self.max_displacement = self.force * self.l**3 / (3 * self.E * self.moment_of_inertia)
    elif type_of_problem=="clamped-simply-supported" or type_of_problem=="simply-supported-free":
      self.natural_frequencies[0] = 15.4 * self.base_frequency
      self.natural_frequencies[1] = 50.0 * self.base_frequency
      self.natural_frequencies[2] = 104.0 * self.base_frequency
      self.natural_frequencies[3] = 178.0 * self.base_frequency
      self.natural_frequencies[4] = 272.0 * self.base_frequency
    elif type_of_problem=="clamped-clamped" or type_of_problem=="free-free":
      self.natural_frequencies[0] = 22.4 * self.base_frequency
      self.natural_frequencies[1] = 61.7 * self.base_frequency
      self.natural_frequencies[2] = 121.0 * self.base_frequency
      self.natural_frequencies[3] = 200.0 * self.base_frequency
      self.natural_frequencies[4] = 296.0 * self.base_frequency


  def average_equilibrium_state(self, lower_indices, higher_indices, time_fraction=0.5, x_max=40, y_min
      = -1.0, res_axis=None, res_name="resolution axis [mm]"):
    self.lower_indices = lower_indices
    self.higher_indices = higher_indices

    from_fraction = 1- time_fraction
    matplotlib.rcParams.update({"figure.figsize": (6.15, 4.15)})

    for dataset, start_id, end_id, label, colour in zip(self.dataset_array, self.lower_indices, self.
        higher_indices, self.label_name, self.colour_array):
      dataset.initialise_displacement_variables(start_id, end_id, time_fraction)

      plt.plot(dataset.x_avg, dataset.z_avg, label=label, color=colour)
      plt.fill_between(dataset.x_avg, dataset.z_s_down, dataset.z_s_up, color=colour, alpha=0.2)

    # calculate the actual displacement
    dx = self.l - dataset.x_avg
    th_displacement = -(1./6.) * self.force / (self.E * self.moment_of_inertia) * (dx**3 - 3 * dx *
        self.l**2 + 2* self.l**3)

    plt.plot(dataset.x_avg, th_displacement, color="k", linestyle="--", label="Equilibrium")

    plt.legend()
    plt.xlabel(f"x coordinate [${dataset.unit_L.units.latex_repr}$]")
    plt.ylabel(f"z displacement [${dataset.unit_L.units.latex_repr}$]")
    plt.xlim(0,x_max)
    plt.ylim(y_min,0)
    plt.savefig(f"{self.save_dir}/average_equilibrium_state.pdf")
    plt.close()

    if res_axis is None:
      return

    dx_highres = dx
    z_avg_highres = dataset.z_avg
    x_avg_highres = dataset.x_avg
    th_displacement_highres = th_displacement

    chi_squared = np.zeros(len(res_axis))
    chi_squared_fine = np.zeros(len(res_axis))
    chi_squared_fine2 = np.zeros(len(res_axis))

    for index, (dataset, start_id, end_id, label, colour) in enumerate(zip(self.dataset_array, self.
        lower_indices, self.higher_indices, self.label_name, self.colour_array)):

      x_avg = dataset.x_avg
      z_avg = dataset.z_avg

      dx = self.l - x_avg
      th_displacement =-(1./6.) * self.force / (self.E * self.moment_of_inertia) * (dx**3 - 3 * dx *
          self.l**2 + 2* self.l**3)

      f = np.interp(x_avg_highres[1:], x_avg, z_avg) * z_avg.units

      chi_squared[index] = np.sum((z_avg[1:] - th_displacement[1:])**2) / len(z_avg[1:])
      chi_squared_fine[index] = np.sum((f - z_avg_highres[1:])**2) / len(f)
      chi_squared_fine2[index] = np.sum((f - th_displacement_highres[1:])**2) / len(f)

    plt.plot(res_axis, chi_squared, "o-", label="Resolution nodes, Chi(theoretical)", linestyle="--",
        color="red")
    plt.plot(res_axis, chi_squared_fine, "o-", label="Highres nodes, Chi(highres)", color="k")
    plt.plot(res_axis, chi_squared_fine2, "o-", label="Highres nodes, Chi(theoretical)", linestyle="--"
        , color="k")
    plt.xlim(np.min(res_axis), np.max(res_axis))
    plt.ylim(1e-7,3e1)
    plt.legend()
    chi_squared_unit = dataset.unit_L**2
    plt.xlabel(res_name)
    plt.ylabel(f"reduced chi-squared [${chi_squared_unit.units.latex_repr}$]")
    plt.xscale("log")
    plt.yscale("log")
    plt.savefig(f"{self.save_dir}/reduced_chi_squared.pdf")
    plt.close()

  def get_fit_parameters(self, index_to_consider, res_axis=None, res_name="Numerical resolution [mm]"):

    dataset = self.dataset_array[0]

    self.fitted_parameters = {}

    self.fitted_parameters["natural_frequencies"] = {}
    self.fitted_parameters["natural_frequencies"]["n=0"] = np.zeros(len(index_to_consider)) * unyt.Hz
    self.fitted_parameters["natural_frequencies"]["n=1"] = np.zeros(len(index_to_consider)) * unyt.Hz
    self.fitted_parameters["natural_frequencies"]["n=2"] = np.zeros(len(index_to_consider)) * unyt.Hz
    self.fitted_parameters["natural_frequencies"]["n=3"] = np.zeros(len(index_to_consider)) * unyt.Hz
    self.fitted_parameters["natural_frequencies"]["n=4"] = np.zeros(len(index_to_consider)) * unyt.Hz
```

```
        self.fitted_parameters["natural_frequencies"]["fitted n=0"] = np.zeros(len(index_to_consider)) * unyt.Hz

        self.fitted_parameters["amplitude"] = np.zeros(len(index_to_consider)) * dataset.unit_L
        self.fitted_parameters["amplitude error"] = np.zeros(len(index_to_consider)) * dataset.unit_L

        self.fitted_parameters["decay time"] = np.zeros(len(index_to_consider)) * unyt.s
        self.fitted_parameters["decay time error"] = np.zeros(len(index_to_consider)) * unyt.s

        self.fitted_parameters["phase offset"] = np.zeros(len(index_to_consider))
        self.fitted_parameters["phase offset error"] = np.zeros(len(index_to_consider))

        for index, (dataset, node_id, label, colour) in enumerate(zip(self.dataset_array, index_to_consider, self.label_name, self.colour_array)):
          time = dataset.time()
          # possible mask, now we just use a dummy mask
          mask = (time > 0.0 * unyt.ms)
          # get the dt
          dt = (time[mask][-1] - time[mask][0])/len(time[mask])

          z = dataset.nodes[f"{node_id:d}"]["dz"]

          mean_displacement = -np.average(z[mask])

          z += mean_displacement

          # based on https://pythontic.com/visualization/signals/fouriertransform_fft
          samplingFrequency = len(time[mask])
          samplingInterval = 1. / samplingFrequency
          samplingInterval = dt
          samplingFrequency = 1./ samplingInterval

          fourierTransform = fft.fft(z[mask])/len(z[mask])
          fourierTransform = fourierTransform[range(int(len(z[mask])/2))]

          tpCount = len(z[mask])
          values = np.arange(int(tpCount/2))
          timePeriod = tpCount/ samplingFrequency
          frequencies = 2*np.pi*values/timePeriod

          abs_fourier = np.abs(fourierTransform)
          dataset.set_FFT_spectrum(frequencies, abs_fourier)

          plt.plot(frequencies, abs_fourier, label=label, color=colour)

        for freq in self.natural_frequencies:
          plt.axvline(x=freq, color="k", linestyle="--")

        plt.xscale("log")
        plt.yscale("log")
        plt.ylim(1e-5,1e2)
        plt.xlim(frequencies[1]*30., frequencies[-1])
        plt.legend()
        plt.xlabel(f"frequency [${dataset.unit_Freq.units.latex_repr}$]")
        plt.ylabel(f"Fourier strength [Arbirary unit]")
        plt.savefig(f"{self.save_dir}/frequency_plot.pdf")
        plt.close()

        for index, (dataset, node_id, label, colour) in enumerate(zip(self.dataset_array, index_to_consider, self.label_name, self.colour_array)):
          frequencies = dataset.frequencies
          abs_fourier = dataset.absolute_FFT

          frequencies.convert_to_units("Hz")
          plt.plot(frequencies, abs_fourier, label=label, color=colour)

        for freq in self.natural_frequencies:
          freq.convert_to_units("Hz")
          plt.axvline(x=freq, color="k", linestyle="--")
        plt.xscale("log")
        plt.yscale("log")
        plt.ylim(1e-5,1e2)
        plt.xlim(frequencies[1]*30., frequencies[-1])
        plt.legend()
        plt.xlabel(f"frequency [${frequencies.units.latex_repr}$]")
        plt.ylabel(f"Fourier strength [Arbirary unit]")

        plt.savefig(f"{self.save_dir}/frequency_plot_Hz.pdf")
        plt.close()

        # find the peaks in the frequencies
        for index, (dataset, node_id, label, colour) in enumerate(zip(self.dataset_array, index_to_consider, self.label_name, self.colour_array)):
          frequencies = dataset.frequencies
          abs_fourier = dataset.absolute_FFT
          frequencies.convert_to_units("Hz")

          natural_frequencies = self.natural_frequencies
          natural_frequencies.convert_to_units("Hz")

          for i, nat_freq in enumerate(natural_frequencies):
            mask = (frequencies < 1.5*nat_freq) & (frequencies > 0.75*nat_freq)

            self.fitted_parameters["natural_frequencies"][f"n={i:d}"][index] = frequencies[mask][np.argmax(abs_fourier[mask])]


        if res_axis is None:
          return

        colour_array_nat_freq = plt.cm.plasma(np.linspace(0,1,len(natural_frequencies)+1))

        for i in range(0,len(natural_frequencies)):

          plt.plot(res_axis, self.fitted_parameters["natural_frequencies"][f"n={i:d}"], "o-", label=f"n={i:d}", color=colour_array_nat_freq[i])

          plt.axhline(y=natural_frequencies[i], color="k", linestyle="--")

        plt.xscale("log")
        plt.yscale("log")
```

```
456       plt.xlim(np.min(res_axis), np.max(res_axis))
457       plt.legend()
458       plt.ylabel(f"frequency [${frequencies.units.latex_repr}$]")
459       plt.xlabel(res_name)
460
461       plt.savefig(f"{self.save_dir}/natural_frequency_convergence.pdf")
462       plt.close()
463
464       # Initial value and boundary values guesses
465       # To be honest, this is a bit of feeling and is not completely
466       # obvious from any viewpoint because fitting is a bit random
467       A_guess = self.max_displacement
468       A_low = 0.9 * A_guess
469       A_upper = 2 * self.max_displacement
470
471       t_decay = 1e3 * unyt.s
472       t_decay_low = np.ones(len(index_to_consider)) * 1e-2 * t_decay
473       t_decay_low[1] *= 4
474       t_decay_upper = 1e5 * t_decay
475
476       omega = natural_frequencies[0]
477       omega_low = 0.9 * omega
478       omega_upper = 1.1 * omega
479       p0 = [A_guess.value, t_decay.value, omega.value, np.pi/2.]
480       phi_low = np.ones(len(index_to_consider)) * 0.9 * np.pi/2
481       phi_low[1] = 0.
482
483
484       for index, (dataset, node_id, label, colour) in enumerate(zip(self.dataset_array, index_to_consider
              , self.label_name, self.colour_array)):
485
486         time=dataset.time()
487         z = dataset.nodes[f"{node_id:d}"]["dz"]
488         mean_displacement = -np.average(z)
489         z += mean_displacement
490         time.convert_to_units("s")
491
492         bounds = ([A_low.value, t_decay_low.value[index], omega_low, phi_low[index]],[A_upper.value,
                t_decay_upper.value, omega_upper, 1.1*p0[-1]])
493         popt, pcov = sco.curve_fit(func_disp, time, z, p0=p0, bounds=bounds)
494         self.fitted_parameters["natural_frequencies"]["fitted n=0"][index] = popt[2]
495
496         self.fitted_parameters["amplitude"][index] = popt[0]
497         self.fitted_parameters["amplitude error"][index] = np.sqrt(pcov[0][0])
498
499         self.fitted_parameters["decay time"][index] = popt[1]
500         self.fitted_parameters["decay time error"][index] = np.sqrt(pcov[1][1])
501
502         self.fitted_parameters["phase offset"][index] = popt[3]
503         self.fitted_parameters["phase offset error"][index] = np.sqrt(pcov[3][3])
504         plt.plot(time, z)
505         plt.plot(time, func_disp(time, popt[0]*unyt.mm, popt[1]*unyt.s, popt[2]/unyt.s, popt[3]))
506         plt.savefig(f"{self.save_dir}/test_{index:d}.pdf")
507         plt.close()
508
509       # Now that we stored all the different fitted values we can compare them and check the convergence
510
511       matplotlib.rcParams.update({"figure.figsize": (6.15, 8.15)})
512       fig, axs = plt.subplots(3, sharex=True)
513       base = self.fitted_parameters["natural_frequencies"]["fitted n=0"][-1]
514       axs[0].plot(res_axis, np.abs(base - self.fitted_parameters["natural_frequencies"]["fitted n=0"])/
              self.fitted_parameters["natural_frequencies"]["fitted n=0"], "o-")
515
516       base = self.fitted_parameters["amplitude"][-1]
517       axs[1].plot(res_axis, np.abs(base - self.fitted_parameters["amplitude"])/self.fitted_parameters["
              amplitude"], "o-")
518
519       base = self.fitted_parameters["decay time"][-1]
520       axs[2].plot(res_axis, np.abs(base - self.fitted_parameters["decay time"])/ self.fitted_parameters["
              decay time"], "o-")
521
522       axs[0].set_ylabel("rel. diff. $\\omega_0$")
523       axs[1].set_ylabel("rel. diff. $A$")
524       axs[2].set_ylabel("rel. diff. $t_{\\rm delay}$")
525       axs[2].set_xlabel(res_name)
526       axs[0].set_xscale("log")
527       axs[0].set_ylim(1e-5,1e0)
528       axs[1].set_ylim(1e-5,1e0)
529       axs[2].set_ylim(1e-5,3e7)
530       for ii in range(0,3):
531         axs[ii].tick_params(axis="both", which="both", direction="in", bottom=True, top=True, right=True,
                left=True)
532         axs[ii].set_yscale("log")
533       plt.subplots_adjust(wspace=0, hspace=0)
534
535       plt.savefig(f"{self.save_dir}/convergence_of_properties.pdf", bbox_inches="tight")
536       plt.close()
537
538
539
540
541   def func_disp(t, A, t_decay, omega, phi):
542     return A * np.sin(omega * t + phi) * np.exp(- t / t_decay)
543
544
545
546   if __name__ == "__main__":
547     directory = ["CantileverBeam40mmx10mmRes5mm", "CantileverBeam40mmx10mmRes2.5mm", "
            CantileverBeam40mmx10mmRes1.25mm", "CantileverBeam40mmx10mmRes0.625mm", "
            CantileverBeam40mmx10mmRes0.3125mm", "CantileverBeam40mmx10mmRes0.15625mm"]
548     base_name = ["cantileverbeam40mmx10mmres5mm", "cantileverbeam40mmx10mmres2.5mm", "
            cantileverbeam40mmx10mmres1.25mm", "cantileverbeam40mmx10mmres0.625mm", "
            cantileverbeam40mmx10mmres0.3125mm", "cantileverbeam40mmx10mmres0.15625mm"]
549
550     label_name = ["5 mm", "2.5 mm", "1.25 mm", "0.625 mm", "0.3125 mm", "0.15625 mm"]
551     res_var = [5, 2.5, 1.25, 0.625, 0.3125, 0.15625]
552
553     index_to_consider = np.array([18, 51, 165, 585, 2193, 8481])
554
555     make_vertical_displacement_plot(directory, base_name, index_to_consider, label_name, colour_array)
```

C Radioss file creator

To create RADIOSS starter files for the studied problems a module was created that effectively creates the model and other input in the OPENRADIOSS input format. Additionally, this allows for an efficient way of creating different meshes with different numerical resolutions.

```python
#!/usr/bin/env python3
import numpy as np
import unyt
from unyt import mm, cm, m, kg, g, s
import yaml
from yaml.loader import SafeLoader
import sys


class createRadiossFile:
    def __init__(self, yaml_file_name):
        """! initialise the createRadiossFile class

            @param self the instance of the class
            @param yaml_file_name  the yaml file name

            @return initialised class
            """
            self.yaml_file_name = yaml_file_name

            # Load the YAML file
            self.load_YAML_file()

            # initialise the node grid
            self.initialise_nodes()

    def print_rad_file(self):
        """! Print the rad file to the screen

            @param self the instance of the class

            @return   Print the rad file to the screen
            """
            self.begin_section()

            if self.material_number == 1:
                self.mat_law_no1()
            elif self.material_number == 2:
                self.mat_law_no2()

            if self.type_of_element == "shell":
              self.create_sheet_of_nodes()
            elif self.type_of_element == "beam":
              self.create_line_of_nodes()

            self.create_boundary_condition(1, "111 111", 2)

            if self.type_of_element == "shell":
              self.group_of_nodes("Corresponding boundary nodes", 2, self.ID_nodes[:, 0])
            elif self.type_of_element == "beam":
              self.group_of_nodes("Corresponding boundary nodes", 2, [self.ID_nodes[0]])

            self.create_boundary_condition(2, "000 000", 3)

            if self.type_of_element == "shell":
              self.group_of_nodes("Corresponding boundary nodes", 3, self.ID_nodes[:, -1])
              self.create_quad_shell_elements()
              self.create_mov()
              self.write_shell_properties(1)
            elif self.type_of_element == "beam":
              self.group_of_nodes("Corresponding boundary nodes", 3, [self.ID_nodes[-1]])
              self.create_beam_elements()
              self.write_beam_properties(1)

            # Write a function to the screen
            x = np.zeros(2)
            x[1] = 1e30
            y = np.ones(2)
            self.write_function(1, x, y)

            if self.type_of_element == "shell":
              self.write_force_shell()
            elif self.type_of_element == "beam":
              self.write_force_beam()

            # Define the part
            self.print_section_rad_file(" Parts")
            print("/PART/3")
            print("# part title")
            print("Our_part")
            print("#prop_ID | mat_ID  |subset_ID| virtual thickness| ")
            print("         1         2                             ")

            if self.use_damping == True:
                self.damping(5)

            # time outputs for nodes
            self.print_section_rad_file(
                "Input the desired information for the time file output"
            )
            print("/TH/NODE/2")
            print("# title of nodes to follow in depth")
            print("TH_Measuring_Nodes")
            print(
                "# var_ID1|  var_ID2|  var_ID3|  var_ID4|  var_ID5|  var_ID6|  var_ID7|  var_ID8|  var_ID9|
```

```
                    var_ID10|"
        )
        print("       DEF       XYZ         A       DRX       DRY       DRZ")
        print("# node ID| skew_ID or frame_ID")

        # Select the middle nodes for shells and all nodes for beams
        if self.type_of_element == "shell":
          list_of_indices_in_text_file = self.ID_nodes[
              int(np.floor(self.N_elements_width / 2.0)), :
          ]
          skew_ID = 1
        elif self.type_of_element == "beam":
          list_of_indices_in_text_file = self.ID_nodes
          skew_ID = 0

        for current_node_ID in list_of_indices_in_text_file:
            print(f"{current_node_ID:10d}{skew_ID:10d}")

    def load_YAML_file(self):
        """! Load the YAML file to variables in the class

        @param self the instance of the class

        @return nothing
        """
        # open the YAML file itself
        # initialize all the different variables
        with open(self.yaml_file_name) as f:
            self.yaml_data = yaml.load(f, Loader=SafeLoader)

            # read the strings of the units
            self.unit_M_str = self.yaml_data["InternalUnitSystem"]["UnitMass"]
            self.unit_L_str = self.yaml_data["InternalUnitSystem"]["UnitLength"]
            self.unit_T_str = self.yaml_data["InternalUnitSystem"]["UnitTime"]

            # set the correct units in the file
            if self.unit_M_str == "kg":
                self.unit_M = unyt.kg
            elif self.unit_M_str == "g":
                self.unit_M = unyt.g
            else:
                sys.exit("Undefined unit for mass!")

            if self.unit_L_str == "mm":
                self.unit_L = unyt.mm
            elif self.unit_L_str == "m":
                self.unit_L = unyt.m
            elif self.unit_L_str == "cm":
                self.unit_L = unyt.cm
            else:
                sys.exit("Undefined unit for length!")

            if self.unit_T_str == "ms":
                self.unit_T = unyt.ms
            elif self.unit_T_str == "s":
                self.unit_T = unyt.s
            else:
                sys.exit("Undefined unit for time!")

            # set the dimensions of the beam element
            self.thickness = (
                float(self.yaml_data["BeamDimension"]["Thickness"]) * self.unit_L
            )
            self.width = float(self.yaml_data["BeamDimension"]["Width"]) * self.unit_L
            self.length = float(self.yaml_data["BeamDimension"]["Length"]) * self.unit_L

            self.area = self.thickness * self.width
            self.I_ZZ = self.thickness * self.width**3 / 12.
            self.I_YY = self.thickness**3 * self.width / 12.

            # set the zero coordinates
            self.x_0, self.y_0, self.z_0 = self.yaml_data["BeamDimension"][
                "CoordinateSystemZeroPoint"
            ]
            self.x_0 *= self.unit_L
            self.y_0 *= self.unit_L
            self.z_0 *= self.unit_L

            # set the type of element
            self.type_of_element = self.yaml_data["NumericalResolution"][
                "TypeOfElement"
            ]

            self.small_strain_option_flag = (
                    int(self.yaml_data["NumericalResolution"]["SmallStrainOptionFlag"])
            )

            if self.type_of_element == "shell":
                self.shell_element_size = (
                    float(self.yaml_data["NumericalResolution"]["ShellElementSize"])
                    * self.unit_L
                )
                self.vertical_integration_points = int(
                    self.yaml_data["NumericalResolution"]["VerticalIntegrationPoints"]
                )
                self.shell_thickness = self.thickness
                self.shell_four_formulation = int(
                    self.yaml_data["NumericalResolution"]["ShellFourFormulation"]
                )
                try:
                    self.shell_numerical_damping = float(
                        self.yaml_data["NumericalResolution"]["ShellNumericalDamping"]
                    )
                except:
                    self.shell_numerical_damping = 0.015
            elif self.type_of_element == "beam":
                self.beam_element_size = (
                    float(self.yaml_data["NumericalResolution"]["BeamElementSize"])
                    * self.unit_L
                )
                self.beam_membrane_damping = (
```

```
                    float(self.yaml_data["NumericalResolution"]["BeamMembraneDamping"])
                )
                self.beam_flexural_damping = (
                    float(self.yaml_data["NumericalResolution"]["BeamFlexuralDamping"])
                )
                self.beam_formulation_flag = (
                    int(self.yaml_data["NumericalResolution"]["BeamFormulationFlag"])
                )

                self.I_XX = self.beam_element_size * self.width**3 / 12.

            # def write_shell_properties(shell_number, shell_thickness, node_4_formulation=24,
                  number_of_shell_integration_points=5):
            # Read the metadata
            self.run_name = self.yaml_data["MetaData"]["RunName"]
            self.author = self.yaml_data["MetaData"]["Author"]

            # Read the material data
            self.material_number = int(
                self.yaml_data["MaterialProperties"]["LawNumber"]
            )

            # Read general properties of the material
            self.density = float(self.yaml_data["MaterialProperties"]["Density"])
            self.youngs_modulus = float(
                self.yaml_data["MaterialProperties"]["YoungsModulus"]
            )
            self.poisson_ratio = float(
                self.yaml_data["MaterialProperties"]["PoissonRatio"]
            )

            if self.material_number == 2:
                self.yield_stress = float(
                    self.yaml_data["MaterialProperties"]["YieldStress"]
                )
                self.material_input_type_flag = int(
                    self.yaml_data["MaterialProperties"]["InputTypeFlag"]
                )
                self.ultimate_tensile_eng_stress = float(
                    self.yaml_data["MaterialProperties"][
                        "UltimateTensileEngineeringStress"
                    ]
                )
                self.eng_strain_at_UTS = float(
                    self.yaml_data["MaterialProperties"]["EngineeringStrainAtUTS"]
                )
                self.strain_rate_coeff = float(
                    self.yaml_data["MaterialProperties"]["StrainRateCoefficient"]
                )
                self.material_failure_model = self.yaml_data["MaterialProperties"][
                    "FailureModel"
                ]

            self.total_force = -float(self.yaml_data["AppliedForce"]["TotalForce"])
            self.type_of_force = self.yaml_data["AppliedForce"]["TypeOfForce"]

            try:
                self.force_direction = str(self.yaml_data["AppliedForce"]["ForceDirection"])
            except:
                self.force_direction = "Z"

            self.use_damping = self.yaml_data["Damping"]["UseDamping"]

            # if we use damping read the parameters that are used
            if self.use_damping == True:
                self.rayleigh_mass_damping = float(
                    self.yaml_data["Damping"]["RayleighMassDamping"]
                )
                self.rayleigh_stiffness_damping = float(
                    self.yaml_data["Damping"]["RayleighStiffnessDamping"]
                )
                self.start_time = float(self.yaml_data["Damping"]["StartTime"])
                self.end_time = float(self.yaml_data["Damping"]["EndTime"])

    def initialise_nodes(self):
        """! Initialise the nodes inside the class

        @param self the instance of the class

        @return nothing
        """
        if self.type_of_element == "shell":
          self.initialise_nodes_shell()
        elif self.type_of_element == "beam":
          self.initialise_nodes_beam()

    def initialise_nodes_beam(self):
        """! Initialise the nodes inside the class

        @param self the instance of the class

        @return nothing
        """
        self.N_elements_length = int(self.length / self.beam_element_size)

        self.N_nodes = self.N_elements_length + 1
        self.N_shell = self.N_elements_length

        self.ID_nodes = np.arange(2, self.N_nodes + 2)

        self.ID_orientation_node = 1

        self.ID_beam = np.arange(1, self.N_shell + 1)

    def initialise_nodes_shell(self):
        """! Initialise the nodes inside the class

        @param self the instance of the class

        @return nothing
        """
```

```
316             self.N_elements_width = int(self.width / self.shell_element_size)
317             self.N_elements_length = int(self.length / self.shell_element_size)
318 
319             self.N_nodes = (self.N_elements_width + 1) * (self.N_elements_length + 1)
320             self.N_shell = self.N_elements_width * self.N_elements_length
321 
322             self.ID_nodes = np.arange(1, self.N_nodes + 1)
323             self.ID_nodes = self.ID_nodes.reshape(
324                 (self.N_elements_width + 1, self.N_elements_length + 1)
325             )
326 
327             self.ID_shell = np.arange(1, self.N_shell + 1)
328             self.ID_shell = self.ID_shell.reshape(
329                 (self.N_elements_width, self.N_elements_length)
330             )
331 
332         def print_section_rad_file(self, argument_name):
333             """! Prints the argument name comment section
334 
335             @param argument_name  the name in the comment
336 
337             @return prints the comment section
338             """
339             print("##")
340             print("##" + "-" * 98)
341             print(f"## {argument_name}")
342             print("##" + "-" * 98)
343             self.print_numbers()
344 
345         def print_numbers(self):
346             """! Prints the line indents for the different Radioss blocks
347 
348             @param None
349 
350             @return   prints the ident blocks of radioss to the screen
351             """
352             print(
353                 "
                        #---1----|----2----|----3----|----4----|----5----|----6----|----7----|----8----|----9----|---10----|
                        "
354             )
355 
356         def print_header(self):
357             """! Print a comment section with the name of the test and the author
358 
359             @param none
360 
361             @return         Print header information to the screen
362             """
363             print("#RADIOSS STARTER")
364             print("#" * 100)
365             print("# " + f"{self.run_name:<50}" + "#" * 48)
366             print("# " + f"made by: {self.author:<41}" + "#" * 48)
367             comment2 = "generated with the package of Folkert Nobels"
368             print("# " + f"{comment2:<50}" + "#" * 48)
369             print("#" * 100)
370 
371         def begin_section(self):
372             """! Print the begin section with the units to the screen
373 
374             @param none
375 
376             @return         Print the header + begin section to the screen
377             """
378             self.print_header()
379 
380             self.print_section_rad_file("UNIT section")
381             print("/BEGIN")
382             print("# Runname")
383             print("OpenRadioss test case")
384             print("# Input version")
385             print("      2022         0")
386             print("# input mass unit  | input length unit | input time unit   |")
387             print(f"{self.unit_M_str:>20}{self.unit_L_str:>20}{self.unit_T_str:>20}")
388             print("# work mass unit   | work length unit  | work time unit    |")
389             print(f"{self.unit_M_str:>20}{self.unit_L_str:>20}{self.unit_T_str:>20}")
390             print("##")
391 
392         def mat_law_no2(self):
393             """! Print the material law number 2 to the screen
394 
395             @param none
396 
397             @return   Print the information of material law 2 to the screen
398             """
399             self.print_section_rad_file("Material Law No 2. JOHNSON-COOK ELASTOPLASTIC")
400             print("/MAT/PLAS_JOHNS/2")
401             print("# material title")
402             print("Steel_DP600 ")
403             print("# density          |")
404             print(f"{self.density:20.14E}")
405             print("# Young's modulus  | Poisson's ratio   | input type flag   |")
406             print(
407                 f"{self.youngs_modulus:20.16f} {self.poisson_ratio:19.15f}{self.material_input_type_flag:10
                        d}"
408             )
409             print(
410                 "# yield stress     | ultimate tensile  | eng. strain at    | strain rate coeff.|"
411             )
412             print(
413                 "#                  | eng. stress (UTS) | UTS               | c>0,c=0->no strain|"
414             )
415             print(
416                 f"{self.yield_stress:20.13e} {self.ultimate_tensile_eng_stress:19.13e} {self.
                        eng_strain_at_UTS:19.13e} {self.strain_rate_coeff:19.13e}"
417             )
418             print(
419                 "# strain rate coef.|ref.strain rate(SR)| SR flag | SR smoo-|cutoff frequency   | Hardening
                        coeff.  |"
420             )
421             print(
```

```
            "# def=0.0           |                   |         |thing fl.|for SR smoothing    | (
                unloading)       |"
        )
        print(
            "

                "
        )
        print(
            "# temperature coef.|Melting temperature|specific heat per  |Reference tempera- |"
        )
        print(
            "#                  |Tmelt=0>no temp eff| unit volume       |ture de=298K       |"
        )
        print(
            "

                "
        )

        if self.material_failure_model:
            self.failure_model()

    def mat_law_no1(self):
        """! Print the material law number 1 to the screen

        @param none

        @return   Print the information of material law 2 to the screen
        """
        self.print_section_rad_file("Material Law No 1. Purely linearly elastic model")
        print("/MAT/LAW1/2")
        print("# material title")
        print("Steel_DP600 ")
        print("# density          |")
        print(f"{self.density:20.14E}")
        print("# Young's modulus  | Poisson's ratio   |")
        print(f"{self.youngs_modulus:20.14f}{self.poisson_ratio:20.14f}")

    def failure_model(
        self,
        failure_plastic_strain_in_unaxial_tension=0.75,
        comment="Failure criterion of steel",
    ):
        """! Print the failure model to the screen

        @param comment      comment printed before the failure model block

        @return   print the failure criterion to the screen

        """
        self.print_section_rad_file(comment)
        print("/FAIL/BIQUAD/2")
        print("# Failure plastic strain at:")
        dummy_string = " "
        print(
            "# uniaxial compres.| shear             | in unaxial tension| plain strain tens.| biaxial
                tension   |"
        )
        print(
            f"{dummy_string:>20}{dummy_string:>20}{failure_plastic_strain_in_unaxial_tension:20.14f}"
        )
        print("# Damage accumulation parameters: ")
        print(
            "# Ratio of failed  | Material|specific |instability start  |element size factor|reference
                element  |"
        )
        print(
            "# integ. points    |sel. flag|behv.flag|for loc. necking   |identifier         | size, def
                =1.0     |"
        )
        # Do we want to make this also variables in the function?
        print("                                2         2")

    def print_item_string(self):
        """! print item identation to the screen

        @param none

        @return   Print the item identation to the screen
        """
        print(
            "#item_ID1| item_ID2| item_ID3| item_ID4| item_ID5| item_ID6| item_ID7| item_ID8| item_ID9|
                item_ID10|"
        )

    def group_of_nodes(self, comment, number, group_of_nodes):
        """! Create a group of nodes

        @param comment          group of nodes comment
        @param number           number of the group of nodes
        @param group_of_nodes   array of the ID of the nodes

        @return   Print the group to the screen
        """
        self.print_section_rad_file(comment)
        print(f"/GRNOD/NODE/{number:d}")
        print("# node group for example for bounary condtions")
        self.print_item_string()
        print(
            "grnodnode", end=""
        )  # we skip \n because that is included in the for loop

        # loop through the different nodes
        for i in range(0, len(group_of_nodes)):
            # check if number is devisable by 10 because then we want a new line
            if i % 10 == 0:
                print("\n", end="")

            # write the node number to the file
            print(f"{group_of_nodes[i]:10d}", end="")
```

```
524
525             # space because we want to go the next line and not have the next output starts on this line
526             print(" ")
527
528         def create_line_of_nodes(self):
529             """! Create a line of nodes
530
531             @param none
532
533             @return   print the node information to the screen
534             """
535             self.print_section_rad_file("NODES")
536             print("/NODE")
537             print("# node_ID| X coordinate      | Y coordinate      | Z coordinate      |")
538             x = self.x_0
539             y = self.y_0 + self.length
540             z = self.z_0
541             print(f"{self.ID_orientation_node:10d} {x.value:19.6f} {y.value:19.6f} {z.value:19.6f}")
542             for i in range(0, self.N_elements_length + 1):
543               x = self.x_0 + self.length * i / (self.N_elements_length)
544               y = self.y_0
545               z = self.z_0
546               print(f"{self.ID_nodes[i]:10d} {x.value:19.6f} {y.value:19.6f} {z.value:19.6f}")
547
548         def create_sheet_of_nodes(self):
549             """! Create a sheet of nodes
550
551             @param none
552
553             @return   print the node information to the screen
554             """
555             self.print_section_rad_file("NODES")
556             print("/NODE")
557             print("# node_ID| X coordinate      | Y coordinate      | Z coordinate      |")
558             for i in range(0, self.N_elements_width + 1):
559                 for j in range(0, self.N_elements_length + 1):
560                     x = self.x_0 + self.length * j / (self.N_elements_length)
561                     y = self.y_0 + self.width * i / (self.N_elements_width)
562                     z = self.z_0
563                     print(
564                         f"{self.ID_nodes[i,j]:10d} {x.value:19.6f} {y.value:19.6f} {z.value:19.6f}"
565                     )
566
567         def create_boundary_condition(
568             self, boundary_ID, DOF_string, particle_group, skewnr=0
569         ):
570             """! print the boundary condition section to the screen
571
572             @param boundary_ID  ID of this boundary condition
573             @param DOF_string   degrees of freedom string
574             @particle_group     ID of the particle group this should apply to
575             @skewnr (optional)  skew number
576
577             @return print the boundary section to the screen
578             """
579             self.print_section_rad_file(f"Boundary Conditions {boundary_ID:d}")
580             print(f"/BCS/{boundary_ID:d}")
581             print("# boundary condition title")
582             print(f"constraint{boundary_ID:d}")
583             print("#Trarot  | skew_ID | grnd_ID |")
584             print(f"{DOF_string:>10}{skewnr:10d}{particle_group:10d}")
585
586         def create_mov(self):
587             """! print the mov part to the screen
588
589             @param none
590
591             @return print the mov section to the screen
592             """
593             # Define the relative motion
594             self.print_section_rad_file("Frames - Mov")
595             print("/FRAME/MOV/1")
596             print("# frame title")
597             print(" ")
598             ID1 = self.ID_nodes[0, 0]
599             ID2 = self.ID_nodes[0, 1]
600             ID3 = self.ID_nodes[1, 0]
601             print("#node_ID1| node_ID2| node_ID3| dir ID1 ID2 axis (def=X)")
602             print(f"{ID1:10d}{ID2:10d}{ID3:10d}")
603
604         def create_beam_elements(self):
605             """! print the quad shell output for the input ID_shell and ID_nodes
606
607             @param none
608
609             @return print the quad shell section to the screen
610             """
611             self.print_section_rad_file("Quad Shell Elements")
612             print("/BEAM/3")
613             print("# beam ID| node_ID1| node_ID2| node_ID3|")
614             for i in range(0, self.N_elements_length):
615               current_node_IDs = [self.ID_nodes[i], self.ID_nodes[i+1], self.ID_orientation_node]
616               print(f"{self.ID_beam[i]:10d}{current_node_IDs[0]:10d}{current_node_IDs[1]:10d}{
                        current_node_IDs[2]:10d}")
617
618         def create_quad_shell_elements(self):
619             """! print the quad shell output for the input ID_shell and ID_nodes
620
621             @param none
622
623             @return print the quad shell section to the screen
624             """
625             self.print_section_rad_file("Quad Shell Elements")
626             print("/SHELL/3")
627             print(
628                 "#shell ID| node_ID1| node_ID2| node_ID3| node_ID4|         |Orthotropy angle   |shell
                        thickness"
629             )
630             print(
631                 "#                                                        |wrt element skew   |default->
                        specified property"
```

```
            )
            for i in range(0, self.N_elements_width):
                for j in range(0, self.N_elements_length):
                    current_node_IDs = self.ID_nodes[i : i + 2, j : j + 2]
                    current_node_IDs = current_node_IDs.flatten()
                    print(
                        f"{self.ID_shell[i,j]:10d}{current_node_IDs[1]:10d}{current_node_IDs[3]:10d}{
                            current_node_IDs[2]:10d}{current_node_IDs[0]:10d}"
                    )
        def write_beam_properties(self, beam_number):
            """! Print the shell properties of this type of shell

            @param beam_number       ID of beam type

            @return print the beam properties to the screen
            """

            blanc = " "
            self.print_section_rad_file(f"Beam Property Set (pid {beam_number:d})")
            print(f"/PROP/BEAM/{beam_number:d}")
            print("line_1.7")
            print("#         | I_SMSTR |")
            print(f"{blanc:<10}{self.small_strain_option_flag:10d}")
            print("#beam membrane damp|beam flexural damp |")
            print(f"{self.beam_membrane_damping:20.13f}{self.beam_flexural_damping:20.13f}")
            print("# Area              | I_YY              | I_ZZ              | I_XX              |")
            print(f"{self.area.value:20.13f}{self.I_YY.value:20.13f}{self.I_ZZ.value:20.13f}{self.I_XX.
                value:20.13f}")
            print("#OmegaDOF|  I_shear|")
            DOF = "000 000"
            print(f"{DOF:>10}{self.beam_formulation_flag:10d}")

        def write_shell_properties(self, shell_number):
            """! Print the shell properties of this type of shell

            @param shell_number       ID of shell type

            @return print the shell properties to the screen
            """

            blanc = " "
            self.print_section_rad_file(f"Shell Property Set (pid {shell_number:d})")
            print(f"/PROP/SHELL/{shell_number:d}")
            print("# Shell properties title")
            print("sheet_1.7")
            print(
                "# 4 node |shellsmall|3 node  |drilling | pinch dof         | ratio of through thickness"
            )
            print(
                "# element|strain    |element |dof stiff|                   | integration points that must"
            )
            print(
                "#      -formulation-         |-ness    |                   | fail bfore the element is
                delated"
            )
            print(f"{self.shell_four_formulation:10d}                              ")

            print(
                "# Shell membrane    | Shell out-of-    | shell rotation    | shell membrane    | shell
                    numerical   |"
            )
            print(
                "# hourglass coeff.  | plane hourglass  | hourglass coeff.  | damping           | damping
                            |"
            )
            print(
                f"{blanc:<20}{blanc:<20}{blanc:<20}{blanc:<20}{self.shell_numerical_damping:20.14f}"
            )

            print(
                "# number of integra-| shell thickness  | Shear factor      | Shell resultant   | Shell
                    plane stress|"
            )
            print(
                "#tion points through|                  |                   | stress calculation|
                    plasticity flag   |"
            )
            print(
                "#the thickness <10  |                  |                   |                   |
                                    |"
            )

            print(
                f"{self.vertical_integration_points:10d}{blanc:<10}{self.shell_thickness.value:20.14f}
                                                        1         1"
            )

        def write_function(self, function_ID, x, y):
            """! Write a function that can be called inside the radioss file to the screen

            @param function_ID  function ID (is used to call this function in other code blocks)
            @param x            array of x values
            @param y            array of y values

            @return write the function to the screen
            """
            self.print_section_rad_file(f"Function {function_ID:d}")
            print(f"/FUNCT/{function_ID:d}")
            print("Load")
            print("#" + " " * 18 + "X" + " " * 19 + "Y")
            for xi, yi in zip(x, y):
                print(f"{xi:20.13e}{yi:20.13e}")

        def write_force_beam(
            self,
            direction="Z",
            group_ID_edge=1,
            cload_ID_edge=3,
        ):
            """! Write the force with its group nodes

```

```
734             @param direction (optional)   direction of the force
735             @param group_ID_edge (optional)   ID of the group of particles in the corner/edge
736             @param cloud_ID_edge (optional)   ID of cload on the edge/corner particles
737 
738             @return write the imposed force to the screen with the different nodel group that are required
                    for this force
739             """
740             if self.type_of_force == "line force":
741               # We only need the node at the very edge
742               nodes_boundary = [self.ID_nodes[-1]]
743 
744               # Calculate the total force
745               force = self.total_force
746               # print information about the line testing
747               self.print_section_rad_file(
748                   "Line loading for test"
749               )
750               print(f"/CLOAD/{cload_ID_edge}")
751               print("# title of the imposed load")
752               print("imposed_load")
753               print(
754                   "#fct_IDT | Dir     |skew_ID | sens_ID | grnd_ID  |         | Ascale_x          |
                          Fscale_y          |"
755               )
756               blanc = " "
757 
758               fscale = force
759               print(
760                   f"{1:10d}{direction:>10}{blanc:>10}{blanc:>10}{group_ID_edge:10d}{blanc:>10}{1.0:20.16f}{
                          fscale:20.16f}"
761               )  # {" ":<10}{" ":<10}{" ":<10}{1.0:10.6f}{1.0:10.6f}")
762 
763               # print the group of nodes of interest
764               self.group_of_nodes(
765                   "Particles for load in the cente at the boundary",
766                   group_ID_edge,
767                   nodes_boundary,
768               )
769 
770             elif self.type_of_force == "area force":
771               error(0)
772 
773         def write_force_shell(
774             self,
775             direction="Z",
776             group_ID_centre=1,
777             group_ID_edge=4,
778             cload_ID_centre=3,
779             cload_ID_edge=4,
780         ):
781             """! Write the force with its group nodes
782 
783             @param direction (optional)   direction of the force
784             @param group_ID_edge (optional)   ID of the group of particles in the corner/edge
785             @param cloud_ID_edge (optional)   ID of cload on the edge/corner particles
786 
787             Illustration of the difference between corner/edge and centre particles
788             ---------#--# <--- corner particle --> --#--
789                      |      /edge                    |
790             ---------#--# <--- centre particle --> --#--
791                      |                               |
792             ---------#--# <--- centre particle --> --#--
793                      |                               |
794             ---------#--# <--- corner particle --> --#--
795                            /edge
796 
797             @return write the imposed force to the screen with the different nodel groups that are required
                    for this force
798             """
799             if self.type_of_force == "line force" and self.force_direction=="Z":
800                 # Determine the nodes that are at the boundary
801                 nodes_second_rigid = self.ID_nodes[:, -1:].flatten()
802 
803                 # Determine the nodes that are at the corner and not at the corner
804                 nodes_not_edge = nodes_second_rigid[1:-1]
805                 nodes_edge = [nodes_second_rigid[0], nodes_second_rigid[-1]]
806 
807                 # unit of force
808                 # corners have half the force of the edge because they are just attached to
809                 # a single shell element instead of two
810                 force_corner_node = self.total_force / (
811                     len(nodes_edge) + 2 * len(nodes_not_edge)
812                 )
813                 force_boundary_node = 2 * force_corner_node
814 
815                 if len(nodes_not_edge) < 1:
816                     elements_in_centre = False
817                 else:
818                     elements_in_centre = True
819 
820                 if elements_in_centre:
821                     # print information about the line testing
822                     self.print_section_rad_file(
823                         "Line loading for test (elements in centre of load)"
824                     )
825                     print(f"/CLOAD/{cload_ID_centre}")
826                     print("# title of the imposed load")
827                     print("imposed_load")
828                     print(
829                         "#fct_IDT | Dir     |skew_ID | sens_ID | grnd_ID  |         | Ascale_x          |
                                Fscale_y          |"
830                     )
831                     blanc = " "
832 
833                     fscale = force_boundary_node
834                     print(
835                         f"{1:10d}{direction:>10}{blanc:>10}{blanc:>10}{group_ID_centre:10d}{blanc
                                :>10}{1.0:20.16f}{fscale:20.16f}"
836                     )  # {" ":<10}{" ":<10}{" ":<10}{1.0:10.6f}{1.0:10.6f}")
837 
838                     # print the group of nodes of interest
```

```
                self.group_of_nodes(
                    "Particles for load in the cente at the boundary",
                    group_ID_centre,
                    nodes_not_edge,
                )

            self.print_section_rad_file("Line loading for test (corner)")
            print(f"/CLOAD/{cload_ID_edge}")
            print("# title of the imposed load")
            print("imposed_load")
            print(
                "#fct_IDT | Dir     |skew_ID | sens_ID | grnd_ID  |         | Ascale_x          |
                    Fscale_y          |"
            )
            blanc = " "
            direction = "Z"

            fscale = force_corner_node
            print(
                f"{1:10d}{direction:>10}{blanc:>10}{blanc:>10}{group_ID_edge:10d}{blanc:>10}{1.0:20.16f
                    }{fscale:20.16f}"
            )  # {" ":<10}{" ":<10}{" ":<10}{1.0:10.6f}{1.0:10.6f}")

            self.group_of_nodes(
                "Particles for load at the edge", group_ID_edge, nodes_edge
            )
        elif self.type_of_force == "line force" and self.force_direction=="Y":
            # Determine the nodes that are at the boundary
            nodes_second_rigid = self.ID_nodes[:, -1:].flatten()

            ID_of_node = nodes_second_rigid[0]

            # get the applied force
            force = -self.total_force

            self.print_section_rad_file("point loading for test")
            print(f"/CLOAD/{cload_ID_edge}")
            print("# title of the imposed load")
            print("imposed_load")
            print(
                "#fct_IDT | Dir     |skew_ID | sens_ID | grnd_ID  |         | Ascale_x          |
                    Fscale_y          |"
            )
            blanc = " "
            direction = "Y"
            fscale = force
            print(
                f"{1:10d}{direction:>10}{blanc:>10}{blanc:>10}{group_ID_edge:10d}{blanc:>10}{1.0:20.16f
                    }{fscale:20.16f}"
            )  # {" ":<10}{" ":<10}{" ":<10}{1.0:10.6f}{1.0:10.6f}")

            self.group_of_nodes(
                "Particles for load at the edge", group_ID_edge, [ID_of_node]
            )

        elif self.type_of_force == "area force" and self.force_direction=="Z":

            # Determine the nodes that have 4 shells
            nodes_four_shells = ID_nodes[1:-1, 1:-1].flatten()

            # Determine the nodes that have two shells, these are 3 boundaries, because
            # for one boundary we impose a boundary condition.
            nodes_two_shells = ID_nodes[:, -1].flatten()[1:-1]
            nodes_two_shells = np.append(nodes_two_shells, ID_nodes[0, 1:].flatten())
            nodes_two_shells = np.append(nodes_two_shells, ID_nodes[-1, 1:].flatten())

            # determine the two nodes that have only 1 shell element
            nodes_one_shell = np.array([ID_nodes[:, -1][0], ID_nodes[:, -1][1]])

            force_one_shell = force_total / (
                len(nodes_one_shell)
                + 2 * len(nodes_two_shells)
                + 4 * len(nodes_four_shells)
            )
            force_two_shell = 2.0 * force_one_shell
            force_four_shell = 4.0 * force_one_shell

            # print information about the line testing
            print_section_rad_file("Surface force for the central nodes")
            print(f"/CLOAD/{cload_ID_centre}")
            print("# title of the imposed load")
            print("imposed_load")
            print(
                "#fct_IDT | Dir     |skew_ID | sens_ID | grnd_ID  |         | Ascale_x          |
                    Fscale_y          |"
            )
            blanc = " "

            fscale = force_four_shell
            print(
                f"{1:10d}{direction:>10}{blanc:>10}{blanc:>10}{group_ID_centre:10d}{blanc
                    :>10}{1.0:20.16f}{fscale:20.16f}"
            )  # {" ":<10}{" ":<10}{" ":<10}{1.0:10.6f}{1.0:10.6f}")

            # print the group of nodes of interest
            group_of_nodes(
                "Particles for load in the cente at the boundary",
                group_ID_centre,
                nodes_four_shells,
            )

            # Force on edge
            print_section_rad_file("Line loading for test (edge)")
            print(f"/CLOAD/{cload_ID_edge}")
            print("# title of the imposed load")
            print("imposed_load")
            print(
                "#fct_IDT | Dir     |skew_ID | sens_ID | grnd_ID  |         | Ascale_x          |
                    Fscale_y          |"
            )
            blanc = " "
```

```
            direction = "Z"

            fscale = force_two_shell
            print(
                f"{1:10d}{direction:>10}{blanc:>10}{blanc:>10}{group_ID_edge:10d}{blanc:>10}{1.0:20.16f}{fscale:20.16f}"
            )  # {" ":<10}{" ":<10}{" ":<10}{1.0:10.6f}{1.0:10.6f}")

            group_of_nodes(
                "Particles for load at the edge", group_ID_edge, nodes_two_shells
            )

            group_ID_corner = group_ID_edge + 10
            # Force on corner
            print_section_rad_file("Line loading for test (edge)")
            print(f"/CLOAD/{cload_ID_edge}")
            print("# title of the imposed load")
            print("imposed_load")
            print(
                "#fct_IDT | Dir     |skew_ID | sens_ID | grnd_ID  |          | Ascale_x          |          Fscale_y          |"
            )
            blanc = " "
            direction = "Z"

            fscale = force_one_shell
            print(
                f"{1:10d}{direction:>10}{blanc:>10}{blanc:>10}{group_ID_corner:10d}{blanc:>10}{1.0:20.16f}{fscale:20.16f}"
            )  # {" ":<10}{" ":<10}{" ":<10}{1.0:10.6f}{1.0:10.6f}")

            group_of_nodes(
                "Particles for load at the edge", group_ID_corner, nodes_one_shell
            )

    def damping(self, group_ID):
        self.group_of_nodes(
            "All particles should be damped", group_ID, self.ID_nodes.flatten()
        )

        self.damping_parameters(group_ID)

    def damping_parameters(self, group_ID, skew_ID=None, damp_ID=1, title="damp_title"):
        """! Create a damping section

        @param alpha      Mass damping coefficient used for all DOF
        @param beta       Stiffness damping coefficient used for all DOF
        @param group_ID   The group ID on which the damping is applied
        @param start_time Start time of the damping
        @param end_time   Stop time of the damping
        @param skew_ID (optional) skew identifier
        @param damp_ID (optional) damping identifier
        @param title   (optional) title name of the damping

        @return   Print the damping section to the screen
        """

        self.print_section_rad_file("Damping section")
        print(f"/DAMP/{damp_ID:d}")
        print("# Title of damping section")
        print(f"{title}")
        print(
            "# Mass damp coeff. |stiffness dampcoeff| grnd_ID | skew_ID | start time        | end time          |"
        )
        skew_string = ""
        if skew_ID is not None:
            skew_string = f"{skew_ID:10d}"
        print(
            f"{self.rayleigh_mass_damping:20.14f}{self.rayleigh_stiffness_damping:20.14f}{group_ID:10d}{skew_string:<10}{self.start_time:20.13e}{self.end_time:20.13e}"
        )


################################################################################
# Read the YAML file

# the YAML file is the second argument after the script name
yaml_file_name = str(sys.argv[1])

radioss_file = createRadiossFile(yaml_file_name)

radioss_file.print_rad_file()
```