NTNU
Norwegian University of
Science and Technology

Progressive collapse of buildings caused by explosion

Fredrik Schjelderup Dalen

Civil and Environmental Engineering
Submission date: June 2016
Supervisor: David Morin, KT

Norwegian University of Science and Technology
Department of Structural Engineering

Department of Structural Engineering
Faculty of Engineering Science and Technology
NTNU- Norwegian University of Science and Technology

ACCESSIBILITY
Open

MASTER THESIS 2016

SUBJECT AREA: Structural Engineering	DATE: 10.06.16	NO. OF PAGES: 115

TITLE:

Progressive collapse of buildings caused by explosion

BY:

Fredrik Schjelderup Dalen

SUMMARY:
Extreme loading may cause failure or damage to structural building members. Redistribution of forces might propagate the local failure into a complete or partial progressive collapse of the building. The prevailing method for analyzing the potential for such collapse is the alternate path method. One structural member is notionally removed to see if the forces are able to find an alternate path. Explosions might cause damage to more than one structural member and produce a structural response not included in an alternate path analysis. A literature review have been conducted to study analysis methods using both the alternate path method and methods incorporating blast loading in the collapse analysis.

It is possible to model building collapse resulting from an explosion with complex models that require large computational force. This thesis have tried to use implicit time integration instead of explicit in order to reduce computational cost, but this was not found beneficial.

Beam elements are effective, computationally effective and are easy to model. Only a small number of studies have used blast loading on beam elements in some way. A steel frame building was analyzed using beam elements with an incident wave interaction in Abaqus to model the blast load. This was compared with a model using shell elements to model the steel sections with Conwep blast loading. The blast loading on the beam elements did not produce a satisfactory response.

RESPONSIBLE TEACHER: Associate Professor David Morin

SUPERVISOR(S): Associate Professor David Morin

CARRIED OUT AT: Department of Structural Engineering

Abstract

Extreme loading may cause failure or damage to structural building members. Redistribution of forces might propagate the local failure into a complete or partial progressive collapse of the building. The prevailing method for analyzing the potential for such collapse is the alternate path method. One structural member is notionally removed to see if the forces are able to find an alternate path. Explosions might cause damage to more than one structural member and produce a structural response not included in an alternate path analysis. A literature review have been conducted to study analysis methods using both the alternate path method and methods incorporating blast loading in the collapse analysis.

It is possible to model building collapse resulting from an explosion with complex models that require large computational force. This thesis have tried to use implicit time integration instead of explicit in order to reduce computational cost, but this was not found beneficial.

Beam elements are effective, computationally effective and are easy to model. Only a small number of studies have used blast loading on beam elements in some way. A steel frame building was analyzed using beam elements with an incident wave interaction in Abaqus to model the blast load. This was compared with a model using shell elements to model the steel sections with Conwep blast loading. The blast loading on the beam elements did not produce a satisfactory response.

Table of Contents

List of Tables

List of Figures

Nomenclature

Abbreviations

AP	Alternative Path
DL	Dead Load
DOF	Degree of Freedom
DOF	Degree of Freedom
FE	Finite Element
LL	Live Load
RC	Reinforced concrete

Variables

$\mathbf{D}$	Nodal displacements matrix
$\mathbf{K}$	Stiffness matrix
$\mathbf{R}$	External load matrix
ρ	Density
ρ_f	Fluid density
c	Damping coefficient
i_S	Positive impulse
i_S^-	Negative Impulse
m	Mass
t_0	Positive phase duration

t_0^-	Negative phase duration
Δt	Time increment
Δt_{cr}	Critical time increment
u	Displacement
$\dot{u}$	$\frac{du}{dt}$, Velocity
$\ddot{u}$	$\frac{d^2u}{dt^2}$, Acceleration
u_{n+i}	Displacement at time step $i\Delta t$
C_A	Drag coefficient of section
E	Young's Modulus
L^e	Characteristic element length
P_0	Ambient pressure
P_r	Peak reflected pressure
P_r^-	Peak reflected negative pressure
P_S0	Peak incident pressure
P_S0^-	Peak incident negative pressure
R	Distance from the explosion
W	TNT-equivalent weight of explosive
Z	Scaled distance $\frac{R}{W^{1/3}}$

Chapter 1

Introduction

Progressive collapse as defined by the American Society of Civil Engineers (ASCE) is "the spread of an initial local failure from element to element, eventually resulting in the collapse of an entire structure or a disproportionately large part of it"[1]. The phenomena first became a topic among researchers and implemented in design codes after a partial collapse of a 22 story residential building, Ronan Point, in London in 1968 [2]. A gas explosion caused an entire corner of the building to collapse killing five and injuring 16 residents. More resent examples include the partial collapse of the Alfred P. Murrah Federal Building in Oklahoma City in 1995, caused by a car bomb [3]. The World Trade Center in New York in 2001, caused by impact of a plane and the subsequent fire [4]. In Norway the Government quarter was bombed with a car bomb in 2011. The 18 story high 'Høyblokka', built in 1958, was extensively damaged but no collapse occurred.

Both the US and the UK have developed detailed guidelines on design against progressive collapse and the Eurocodes also contains some regulations regarding this. The criteria in existing guidelines are mostly threat independent based either on tying forces or by analysis of an alternate path for forces after notional removal of a structural member. The effects on a structure from an explosion is difficult to take fully into account with these approaches. With large computational resources becoming more available, more detailed direct simulation of threat scenarios, including explosions, are becoming possible.

Progressive collapse does not only happen in very tall buildings. According to DoD [5] all buildings with three stories or more should take progressive collapse into account, but the taller and larger the building is, the larger the consequences of a collapse will be. Norway does not have a strong tradition for tall buildings. Excluding masts the tallest structures in Norway are offshore platforms like Troll A (472 m, 169 m above sea level) and towers like the bridge towers of the Hardagner bridge (202 m), Tyholt Tower in Trondheim and Nexans Tower in Halden (both about 120 m). The highest building in Norway is Oslo Plaza with 37 stories at 117 m. Most tall buildings are located in Oslo. The city has more that 100 buildings or 12 stories or more stories.

There are no national restrictions on building height, but the local government usually restricts the height of buildings. Traditionally there has not been much need for tall buildings and height restrictions are often in place for aesthetic reasons. Buildings heights in a city have often been restricted by the height of the church spire. The height of new buildings is a political topic in many Norwegian cities, but more are being build today than before.

According to Krauthammer [6], there has been more progressive collapse research related to concrete structures than steel frames. The most common multi-story buildings in Norway are steel frames and prefabricated concrete structures. Moment stiff frames are uncommon for multi-story buildings and stiffness for steel frames is achieved with truss bracing or walls acting as plates. Composite action between concrete slabs and steel beams or plates are not common [7]. Wood is a very common building material for residential buildings in Norway, but they are usually limited to two or three stories. Recently, higher wooden buildings have been developed. For example a 51 m 14 story residential building in Bergen, which is the worlds tallest wooden building, or student housing blocks in Ås and in Trondheim with respectively eight and nine stories.

This thesis will focus on simulation methods that are able to predict progressive collapse resulting from a blast loading. A review of resent research on both progressive collapse in general, and incorporating blast effects will be presented. Finite element modeling techniques will be studied with a focus on the practicability of incorporating a blast load in a progressive collapse simulation of a steel frame building. The focus will be on computational time, modeling techniques and how to apply the blast load. The goal for this study is not to create a realistic model of an actual structure, but to test different modeling techniques to see if they are able to recreate the physical phenomena. Certain important modeling aspects like material modeling and joint modeling will not be in focus.

Chapter 2

Approach

This thesis has two main parts. One part is a literature review of analyses methods for progressive collapse, blast loading on buildings and a possible combination of these. The other part is trying to create a finite element (FE) model that is able to model progressive collapse caused by a blast load.

2.1 Literature review

In the literature review about 100 different articles, books and reports have been collected. They have mostly been found using the web services of Science Direct and Google Scholar. Search therms used includes among others: *Abaqus, blast, implicit integration, progressive collapse, CONWEP, collapse, nonlinear, steel frame*. In addition to articles and reports, books by Fu [8] and Krauthammer [6] and governmental guidelines by the U.S. Department of Defense and Eurocodes have been studied.

2.2 FEM-modeling

The modeling have been done using Abaqus, by Dessault Systemès [9]. Abaqus is a commercial multi-physics software capable of nonlinear FE analyses. It is cumbersome to create large building geometries with the graphical user-interface in Abaqus. The models where therefore created using the Abaqus Scripting Interface. With this interface, input to Abaqus is given using Python scripts. Python is a free high level general purpose programing language. Since the geometry of the building is regular, it is relatively easy to program the geometry using a script. Other benefits of using a script is the possibility to easy parametrize properties of the model and to automate post processing to generate the desired output. MATLAB by The MathWorks, Inc. was used for further processing and plotting of data. 7.2 contains a Python script and two modules that runs a blast analysis of a steel building using beam elements for the frame and shell for the slabs. All scripts used to generate input, models and processes output are available online at `https://github.com/fsdalen/ProgressiveCollapse`. The analyses were ran on a cluster at NTNU SIMLab, using 8 CPUs for each analysis.

The aim is not to correctly predict collapse of an actual structure, but to study different possibilities of implementing blast loading in progressive collapse analyses, and the practicability of these methods. Aspects like material modeling and bracing of the structure have been greatly simplified. The absolute results of the analyses are not valid by themselves, but differences in response and computational time between different modeling techniques will be studied and reported.

Chapter 3

Theory

3.1 Finite Element Analysis

Finite element analysis is an approximate way of solving a field problem, i.e. solving a distribution of dependent variables. In a structural analysis, the variable is displacement. Other values such as strains and stresses may be derived from the displacements. The structural problem is discretized into elements of a finite size. Displacement is measured at the nodes between the elements and interpolated within the element. In a linear FE analysis the problem consists of finding the stiffness **K** and the external loads **R** of the system and solving the following equation for the displacement **D**.

$$\mathbf{K}\mathbf{D} = \mathbf{R}$$

When doing a nonlinear FEA. **K** and **R** may now be functions of **D** and nonlinear constraints may be imposed.

$$\mathbf{K}(\mathbf{D})\mathbf{D} = \mathbf{R}(\mathbf{D})$$

It is generally not possible to solve this equation directly, it has to be solved incrementally. This greatly increases the computational cost, and the analysis becomes more complex. The principle of superposition is no longer valid. Some possibilities with a nonlinear FE analysis compared to a linear:

- Nonlinear material behavior
- Post buckling analysis
- Nonlinear contact constraints
- Large deformations
- Dynamic response
- Finding the ultimate capacity

3.1.1 Explicit time integration

With implicit solution methods equilibrium is established at each step in the analysis. This allows for large time increments as long as it is possible to establish the correct equilibrium at the next step. Each step is computationally costly because it involves equation solving with inverting the stiffness matrix, and it may need to be done multiple times in order to achieve convergence. Explicit methods on the other hand does not require inverting the stiffness matrix and each step is computationally cheap. It requires a small time increment in order to be stable. This makes it suitable for nonlinear problems where convergence is difficult to obtain with larger time steps.

The method of explicit time integration is here shown for a single degree of freedom (DOF) system, but it may easily be generalized to multi-DOF systems. The equation of motion for a single DOF system with mass m, damping coefficient c, stiffness k, load P, acceleration $\ddot{u}$, velocity $\dot{u}$ and displacement u is

$$m\ddot{u} + c\dot{u} + ku = P \tag{3.1}$$

The common method used for explicit time integration is the central difference method [10]. It uses Taylor series expansion for the displacements u_{n+1} and u_{n-1} about time n. By neglecting terms higher than second order you get the following

$$u_{n+1} = u_n + \Delta t\dot{u}_n + \frac{\Delta t^2}{2}\ddot{u}_n \tag{3.2}$$

$$u_{n-1} = u_n - \Delta t\dot{u}_n + \frac{\Delta t^2}{2}\ddot{u}_n \tag{3.3}$$

By using these approximations it is possible to solve the equation of motion for u_{n+1}. The expression will only contain known values at time n and n-1 and will therefore be explicit. If lumped mass and mass-proportional damping is applied for a multi DOF-system, it is not necessary to invert any matrices and each time step will have very low computational cost. It is often desirable to use both stiffness and mass proportional damping. In order to still achieve low computational cost, the half-step central differences method is used. The velocity term in the equation of motion is left lagging half a step behind

$$m\ddot{u}_n + c\dot{u}_{n-1/2} + ku_n = P_n \tag{3.4}$$

By using these approximations for velocities

$$\dot{u}_{n-1/2} = \frac{1}{\Delta t}(u_n - u_{n-1}) \tag{3.5}$$

$$\dot{u}_{n+1/2} = \frac{1}{\Delta t}(u_{n+1} - u_n) \tag{3.6}$$

and Taylor series expansions for the displacement, the equation of motion may be written as

$$\frac{m}{\Delta t^2}u_{n+1} = P_n - ku_n + \frac{m}{\Delta t^2}(u_n + \Delta\dot{u}_{n-1/2}) - c\dot{u}_{n-1/2} \tag{3.7}$$

When mass lumping is used, the mass matrix becomes diagonal and the computational cost is very low for each time increment.

The explicit method is only conditionally stable. That means that if the time step Δt of each increment is larger than the critical time step Δt_{cr}, the solution will become unstable. The physical interpretation of the critical time step is that information should not propagate between two adjacent nodes within one time step. Δt_{cr} is then proportional to the size of smallest element and inverse proportional to the dilatation wave speed of the material. For a one-dimensional problem

$$\Delta t_{cr} = \frac{L^e}{\sqrt{\frac{E}{\rho}}} \tag{3.8}$$

Where L^e is the characteristic element length and $\sqrt{\frac{E}{\rho}}$ is the dilatation wave speed of the material.

3.2 Blast loading

An explosion is defined as "a sudden, loud and violent release of energy" [11]. The effects on structures from explosions are due to blast pressure, impact of fragments and ground shock waves. Impacts and ground wave can damage buildings, but their effect on the structural response is limited compared to the blast pressure. Unless otherwise specified the theory of this section is from UFC 4-340-02: Structures to Resist the Effects of Accidental Explosions from the US DoD [12].

3.2.1 Categorization of explosions

Explosive material

Explosions can either be mechanical, chemical or nuclear. Mechanical explosions may be the sudden release of confined pressure. A chemical (conventional) explosion is the sudden combustion or detonation of a chemical compound. A nuclear explosion is the result of large amounts of energy released from fusion or fission of atoms. The blast load from nuclear explosions are similar to conventional, except the magnitude, but this thesis focuses on conventional explosions.

A conventional explosion is a stable chemical reaction called a detonation. In a detonation, the reaction always happens at a supersonic rate. Large amounts of gas is produced and this causes the blast pressure. Most solid explosives are high-explosives where all the energy is released in the detonation process. In low explosive solids and gases only part of the energy is released in the detonation, while the rest is a combustion at subsonic speed called a deflagration. Deflagration does not significantly affect the blast pressure because of the big difference in speed.

TNT-equivalence

Data about explosions effects are often presented in relation to the weight of a spherical trinitrotoluene (TNT) charge. In order to relate this to other materials and shapes an effective charge weight may be calculated. The weight of the charge is usually of more importance than the shape, and can be related to TNT-equivalent weight by

$$W_e = \frac{H}{H_{TNT}} W \tag{3.9}$$

where W_e is the effective charge weight. H and H_{TNT} is the heat of the explosion and a TNT-explosion with the same weight W as the charge. Table 3.1 and 3.2 shows examples of TNT-equivalent weights of materials and bomb sizes.

Explosive	TNT-equivalent
TNT	1.00
Liquid nitroglycerin	1.45
C4	1.64
60 % nitroglycerin dynamite	0.6

Table 3.1: TNT-equivalent of some explosives [13]

Type	Effective charge weight (kg TNT-equivalent)
Mail bomb	1
Car bomb	20-50
Small truck (2 tons)	250
Large truck (5 tons)	900

Table 3.2: Typical bomb sizes [1]

Location of explosion

The location of the explosion with regards to the surroundings greatly affects how the blast wave propagates. Explosions can be categorized into confined and unconfined. Unconfined explosion can further be subcategorized:

- *Free air burst*: spherical blast with no amplification of the pressure.
- *Air burst*: the blast wave is reflected against the ground before it hits the structure, causing an amplification of the pressure.
- *Surface burst*: the blast wave is reflected at the source of the explosion causing a hemispherical blast.

3.2.2 Blast pressure

The shock front from a blast travels radially out from the source of the explosion at supersonic speed. The particles behind the shock front have subsonic speed. The pressure at a given distance from the source can be described as in figure 3.1. It consists of a positive phase first, and then a negative phase. There is first a violent increase from the ambient pressure P_0 up to the peak incident pressure P_S0. The pressure then decreases back to the ambient pressure before the negative phase follows. The negative phase is longer, and the peak incident negative pressure is lower than the positive. During the negative phase the particle velocity behind the shock front reverses. The positive and negative impulse, i_S and i_S^-, is the integral of the positive and negative curve respectively. The negative impulse is less important to the structural response and is often neglected, but might not be negligible for flexible structures like steel frames. When the blast wave hits a surface it reflects and the pressure is amplified. The reflected pressure is a function of the incident pressure, the reflective surface and the angle of incident. The red curve in figure 3.1 shows a typical reflected pressure curve.

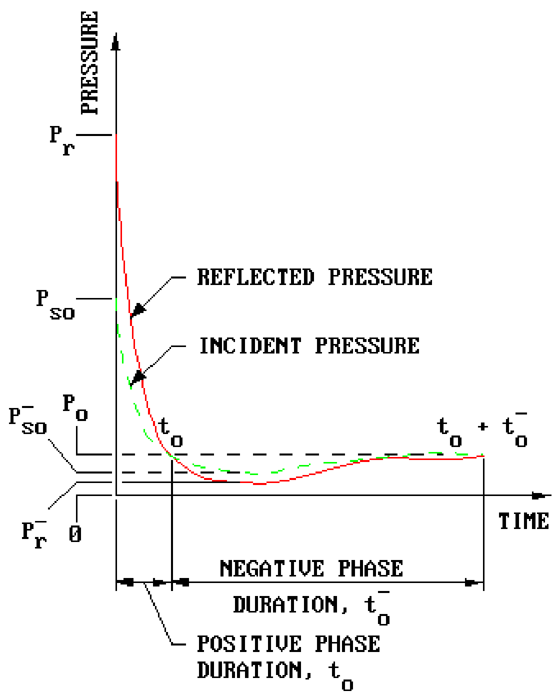


Figure 3.1: Pressure-Time curve [12]

Propagation of the blast wave

For a free air burst explosion, the propagation of the blast wave is purely spherical. For an air burst explosion, a the blast wave reflects off the ground, joins the original blast wave and creates a Mach front as shown in figure 3.2. The Mach front has a vertical front that grows higher further away from the blast source. The pressure over the height of the Mach front is almost uniform and it may be considered a plane wave in the vertical direction. The point at the intersection between the incident wave, the reflected wave and the Mach front is called the triple point. Typical pressure-time curves above and below the triple point is shown in figure 3.3.

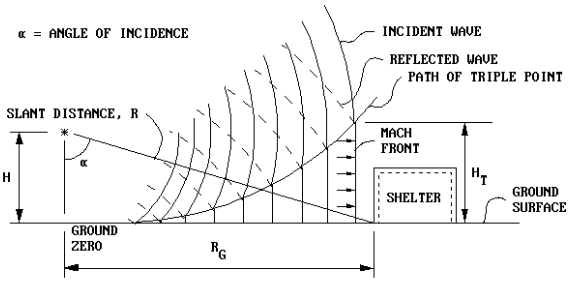


Figure 3.2: Air burst with Mach wave [12]

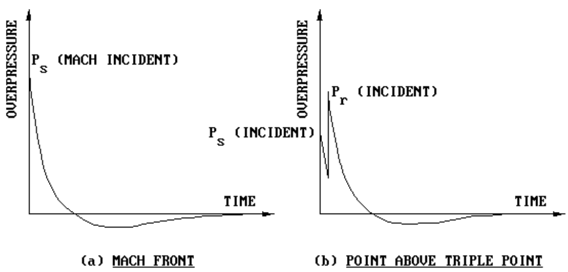


Figure 3.3: Pressure time variation for air burst [12]

When the detonation happens close to the ground, the wave reflecting off the ground joins the original wave from the source combining to one hemispherical wave. For buildings with a low number of stories, the front of the wave might be assumed plane if the standoff distance is large enough. The pressure curve is similar to that of the free air burst except all parameters are magnified.

Blast parameters

There exists analytical and empirical values for a variety of blast parameters. Figure 3.4 shows parameters for a hemispherical surface explosion at sea level. In addition to pressures, impulses and duration of the blast wave, arrival time t_0, velocity of the shock front U and the wave length L_W. The standoff distance is taken into account as a scaled distance Z. It is a function of distance R, and equivalent TNT weight W and is defined by the Hopkinson-Cranz scaling law [14, 15] that is based on conservation of momentum and geometric similarity

$$Z = \frac{R}{W^{1/3}} \tag{3.10}$$

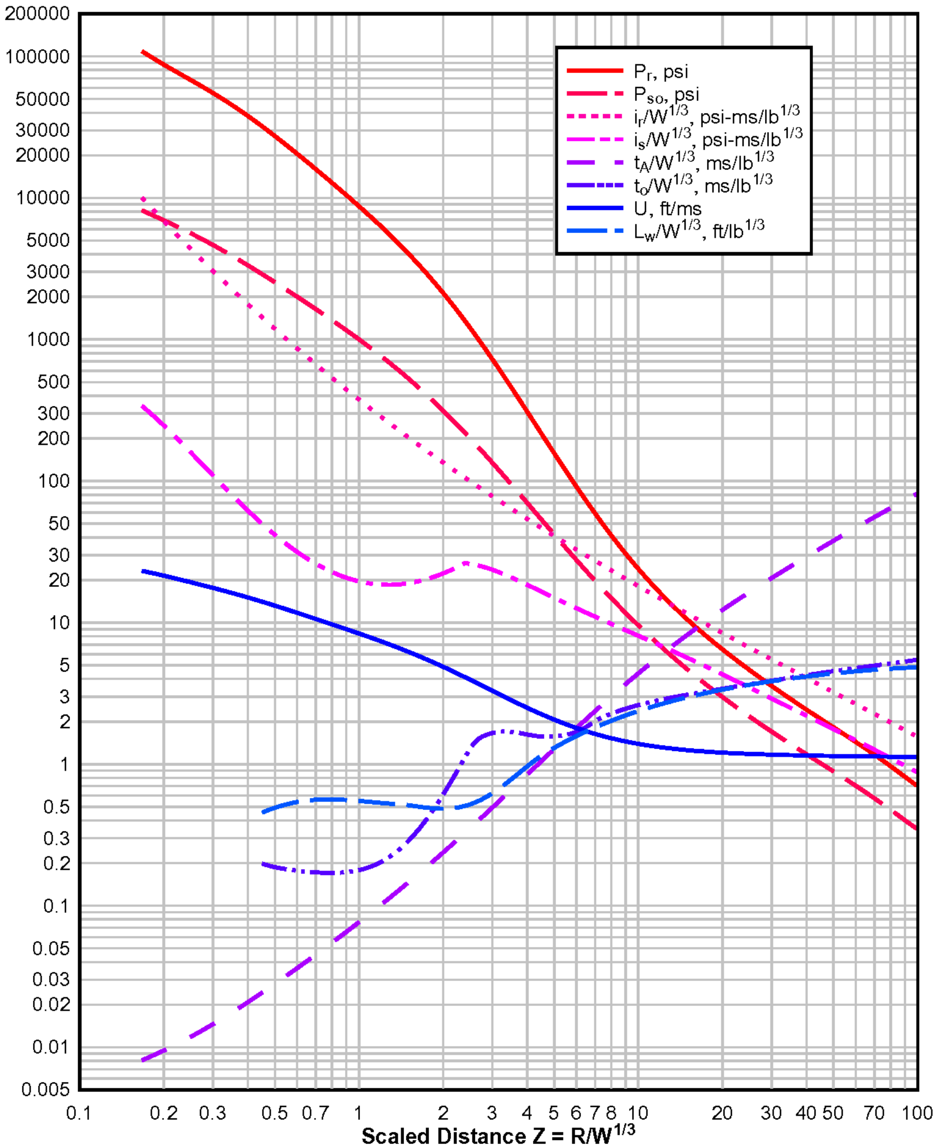


Figure 3.4: Positive blast wave parameters for a hemispherical TNT explosion [12]

Modeling of the time pressure-curve

Given parameters P_{S0} and t_s the simplest way to model the positive pressure-time curve is a linear decay valid from the arrival time until t_0

$$P(t) = P_0 + P_{S0}(1 - \frac{t}{t_0}) \tag{3.11}$$

This equation may be fitted by keeping the correct peak pressure and impulse and varying the phase time t_0. A more complex equation that allows to keep the correct phase time, is the modified Friedlander equation [16]

$$P(t) = P_0 + P_{S0}(1 - \frac{t}{t_0})\mathrm{e}^{\frac{-bt}{t_0}} \tag{3.12}$$

Here, the parameter b is fitted to the other parameters.

Chapter 4

Literature Review

There are literature readily available on both progressive collapse and structural members response to blast loading. But most of the literature on blast loading is limited to structural members or smaller structures. There are only a limited number of studies applying blast loading to multi-story buildings in order to study the collapse potential of the building. In this chapter, design guidelines and studies on progressive collapse will be presented in addition to studies on how to incorporate blast loading in progressive collapse analysis.

4.1 Rules, guidelines and design codes

The UK was the first to create regulations regarding progressive collapse. It started after the Ronan Point accident and the first regulation is from 1972 according to Lui et al. [17]. Today the UK have a paragraph regarding disproportionate collapse in their building regulations with accompanying guidance [18]. In the US much have happens regarding building security after the terrorist attacks on the World Trade Center in 2011. The U.S. Department of Defense have Unified Facilities Criteria for a variety of topics including minimum antiterrorism standards, accidental explosions and progressive collapse [19, 5, 12]. NITS issued a report 'Best Practices for Reducing the Potential for Progressive Collapse in Buildings' in 2007 [20]. Some are publicly available and some are for official use only.

4.1.1 Eurocodes

In Norway the building regulations does not specifically mention progressive collapse or explosions but the Eurocodes includes some of this. Eurocode 0, Basis for Design [21], paragraph 2.1(4) states the following: "A structure shall be designed and executed in such a way that it will not be damaged by events such as: explosion, impact, and the consequences of human to an extent disproportionate to the original cause". Eurocode 1 part 1-7 [22] covers loads caused by accidental actions. It does specifically not cover "external explosions, warfare or terrorist activities", but it does covers strategies for limiting the extent of localized failure caused by an unspecified cause. It states the following approaches for mitigation in section 3.3(2):

a) Designing key elements to withstand a specified accidental load level. This approach is not to be used according to the Norwegian National Annex

b) Alternative path approach. Notional removal of a structural member should not cause collapse of more than 100 m^2 or 15 % (whichever is less).

c) An indirect approach by prescriptive design rules to achieve sufficient tying forces and ductility.

It further gives recommendations on what approaches to use based on consequence classes that takes building type and number of stories. Annex A proposes ways to implement the approaches. It gives specific values for tying forces and it purposes to remove one column, beam or nominal section of a load-bearing wall at a time for the alternate path approach. It states that in the medium consequence class an equivalent static analyses may be adopted, while for the high consequence class nonlinear, dynamic analysis should be considered.

4.2 Approaches to design for Progressive Collapse

According to Marjanishvili [23] the approaches may be divided into one indirect method and two direct methods. The indirect method is a simple method prescribing general design rules to increase the robustness of the structure. This may be done by a prescribing a certain level of tying capabilities and ductility of joints and members. The simplest direct method is designing major structural components to be strong enough to withstand the loads that may initialize collapse. The other direct method is the alternative path (AP) method, where major structural elements are nominally removed and then the structure is analyzed for progressive collapse.

The alternative Path Method is only able to say if the building is able to withstand a nominal member removal. It gives a simple yes or no answer. Attempts have been made at creating a method for quantifying the robustness of a building. Khandelwal and El-Tawil [24] came up with the concept of an Overload Factor and Fascetti et al. [25] expanded this concept to the Local Robustness Evaluation method.

A third direct method would be a collapse analysis based on direct loads such as a blast loading. This method will further be described in section 4.4

4.2.1 Analysis methods for the Alternative Path Method

The progressive collapse analysis may be either linear or nonlinear and either static or dynamic.

Linear Static

In order to account for the dynamic effects when a member is suddenly removed, the loads have to be scaled by a dynamic amplification factor. The factor is often set to two, and may be derived from energy balance as shown by Powell [26]. The method is only valid for simple systems and if an amplification factor of two is used, the method is almost certainly conservative [26]. The disadvantage is that it will often be overly conservative and in more complex systems it might not capture the necessary effects.

Nonlinear Static

This method is often used in seismic analyses and is often called a push-over analysis. For the purpose of progressive collapse it may be called a push-down analysis [25].The vertical loads are increased step by step until failure occurs. The load may be increased globally for the whole structure or locally. The method includes nonlinear effects like catenary effects and ductility. A dynamic amplification factor is necessary to take dynamic effects into account.

Linear Dynamic

With this method the members are removed real-time and the correct dynamic effects are accounted for. First a static analysis is performed. Then a structural member is removed and replaced by the equivalent static forces. The forces are removed and a dynamic analysis is performed. The forces should be removed within 1/10 of the characteristic natural period of the floor [8]. Direct time integration is preferred in order to account for all vibration modes [27]. This method might become nonconservative if the structure exhibits large deformations [23].

Nonlinear Dynamic

Same as linear dynamic, but also accounts for nonlinear effects like nonlinear materials, catenary effects and inelastic buckling. This is the most realistic analysis, but it is also the most difficult to validate and verify and requires the most computational power. Due to the complexity of the analysis, modeling errors are not easily recognized.

4.3 Nonlinear Dynamic analysis

4.3.1 Global modeling approach

When analyzing global models, especially for larger buildings, the level of detail and what assumptions and simplifications to make, that is still practical, depends a lot on computational cost, modeling expertise and what type of results is sought.

The first major consideration is whether to model the whole building or just part of it. Several studies have concluded that for progressive collapse it is insufficient to model a 2D frame [28, 25]. A 2D frame produce collapse at a lower load than 3D because the slabs help to distribute the forces to other frames when columns are removed. A 2D plane frame will therefore become unstable in the plane direction. It is not possible to conclude that a plane 2D frame is conservative either. The reason is that the 2D frame might show only localized collapse in a limited number of bays, while 3D effects might actually cause a complete collapse of the building as showed by Alashker et al. [28].

The next major consideration is the type of elements to use. Floors are mostly modeled using shell elements, but solid elements is also possible. Beams and columns can be modeled using beam elements or shell elements for steel members and solid elements for concrete members. In the study done by Alashker et al. [28] a 10 story steel frame building was modeled in different ways. One model using shell elements for both concrete slabs and the steel members in the frame, the other using beam elements for the steel frame. Part of the frame was verified against experimental bending studies and the building models were analyzed for progressive collapse using the AP method. Both models where able to reproduce the bending experiment and showed similar response in the AP studies. The shell model was able to capture local effects, but the global response was similar. The shell model consisted of almost 800 000 elements while the beam model about 30 0000. The computational time of the shell model was 57 hours and the beam model 1.5 hours. It was concluded that a well calibrated beam element model gave satisfactory results at a much lower computational cost.

Kwasniewski [29] modeled a steel frame building with composite slabs using shell elements for both steel members in the frame and the composite action concrete slabs. The building was eight stories and the model had a total number of 1.1 million elements.

The remaining capacity after column removal was also studied. If collapse did not occur from the column removal, the loading was increased linearly until collapse after vibrations from the column removal had stabilized. This meant running the simulation time for as long as up to 15 seconds. Only a part of the model extending two bays out from the column removed was analyzed. The simulations took up to 19 days using 60 processors in parallel.

For reinforced concrete (RC) buildings, solid elements have also been used globally [30] or locally [31].

4.3.2 Modeling joints

There are a lot of ways to model joint behavior. For steel frames, that are more flexible than RC frames, joint flexibility could have a large impact on the global response of the building. A problem with modeling detailed joints while using explicit time integration is that the elements might become much smaller than for the structural members, decreasing the critical time step. Some methods used in progressive collapse analyses for steel frames are presented here.

Joints in shell models

Assuming the welds of a rigid joint do not fracture, joints can be modeled by taking care that column and beam mesh line up and have them share nodes. If the mesh do not line up, the beam elements can be connected to the surface of the column shells as done by Alashker et al. [28]. For non rigid, shear tab joints Alashker et al. [28] used a single row of shell elements, and varying the section thickness and material properties to achieve the correct behavior of the joint. Kwasniewski [29] modeled bolted end plates using shell elements for the end plates, beam elements for the bolts and single sided contact between the end plate and column. This produced a nonlinear response that matched well with planar bending experiments. A parameter study showed the joint model was sensitive to mesh size of the end plate shells, the failure strain in the bolts and the contact algorithm used.

Joints in beam models

Pinned and fixed joint are fairly simple, by tying the correct DOFs in the beam end and column. Many different approaches are used in the literature for modeling semi-rigid joints beam element frames. Fu [32] used pinned connections while having a composite action concrete slab continuous over the joint so the overall behavior was as a semi-rigid joint. Alashker et al. [28] used a modified beam element at the ends of the beams. The integration points corresponded to the location of bolts in a shear tab joint and the stiffness of the element was adjusted to produce the correct response. The length of the beam element was artificially long compared to the actual joint in order to avoid a very short critical time step. A nonlinear spring element was used to simulate contact between the beam and column for large rotations.

Jeyarajan et al. [33] developed bi- and tri-linear moment-rotation curves for steel frame joints with composite action slabs using Eurocode spring models and simplified joint models in Abaqus. The non linear semi-rigid joint behavior was implemented using an axial and rotational connector element in the building model.

Another modeling consideration frequently addressed in the literature was that of how to model composite action concrete slabs in steel frame buildings. As it is not common practice to use composite action slabs in Norway, this topic was not in focus for this thesis.

4.3.3 Material modeling

Progressive collapse is a highly nonlinear event that requires nonlinear material models. This has not been throughly studied in this thesis. Steel models needs to incorporate plasticity for large deformations and concrete needs to take into account plasticity, reinforcement and cracking. In order to predict collapse, damage have to be taken into account. Two main approaches used in progressive collapse studies were found. Either material damage with element deletion [34, 31] or running the analysis in steps checking the capacity of member in between [33, 32]. The design criteria may be either design strength, rotations or strains. The members exceeding the criteria are removed in subsequent analyses. In blast analyses, dynamic material properties could be implemented in order to take into account large strain rates caused by the blast loading. This has been shown to have significant influence on the global response [33, 35].

4.3.4 Verification of model

In nonlinear analysis it might be difficult to judge the validity of the results without comparing them to results that are assumed correct. Results from a model may either be compared to experimental results, a numerical model that is assumed more correct or analytical results. Some methods used in the literature are reported here.

Luccioni et al. [30] modeled a real concrete building that had partially collapsed after car bomb in Buenos Aires in 1994. The damage the model produced was compared with photos of the building after the attack. Kwasniewski [29] modeled a steel frame structure that was build in the Cardington Large Building Test Facility in the UK for fire tests. Natural frequencies of the model where compared to measurements on the real test building in order to verify mass and stiffness distribution and the flexibility of joints. The mass of the model was also verified against the real test building. Alashker et al. [28] compared a beam element model with a shell element model with a much higher number of elements. Shi et al. [36] verified their purposed analytical SDOF method for blast progressive collapse analysis by creating a global model using solid elements applying the blast load directly to the building. Their purposed method is reported in section 4.4.1.

It is highly impractical to do full scale experiments on progressive collapse and detailed global models are highly computationally costly. Therefore, most often only local parts of the model is verified. Kwasniewski [29] verified joint behavior against experimental Moment-rotation relationships of similar joints and verified a shell element composite action slab by creating a local model with a finer mesh and using solid elements for the concrete. Several others have also used local experimental [32, 33] and numerical [34] verification approaches. There are a number of ways to verify the material models, but this is not a topic for this thesis.

4.4 Blast analysis

The AP method is a good tool for testing general threat independent robustness of a building. When using the AP method usually only one column is removed as required by guidelines and codes [37, 22]. However, this does not take into account the possibility of full or partial damage of multiple columns or the global response of the building to the blast. The global response might include overturning forces, uplift of floors and horizontal displacements. Several studies have concluded that the AP method is not always conservative for blast loading [34, 33]. However, Fu [35] concluded that as long as the blast is not large enough to severely damage beams or shear-off multiple columns, the AP method could be considered sufficient and conservative. This conclusion was based on a study with a package bomb of 15 kg TNT close to a column. The vertical forces of the blast on the floors caused less axial forces in the columns on the floor while higher shear forces compared to an AP approach. This section will present different methods found in the literature on how to incorporate blast loading in progressive collapse analyses.

4.4.1 Applying the blast pressure on a structure

There are several ways to take blast into account. The blast wave itself may be modeling the propagating of the blast wave through the air using Eulerian elements and the interaction between air elements and structural Lagrangian elements.

A much used approach for this is the Conwep loading that is available in Abaqus and LS-Dyna. Conwep loading applies a surface load on shell and solid elements, but is unable to apply load to beam or truss elements. The pressure and propagation of the blast wave is based on unobstructed free air or surface bursts. It calculates the pressure load on a surface based on the charge-weight, distance and angle of incident. [9]

Elsanadedy et al. [34] applied Conwep loading to solid concrete elements on a local column model and then used the results to determine what elements to remove in the global analysis. The global model used beam elements for the structural frame so it was not possible to apply the Conwep load to the frame in LS-Dyna. The global model was ran with the damaged columns removed at the same time as Conwep blast load was applied to the shell elements in the masonry and glass facade. The facade will then transfer loading to the structure, but this approach does not fully capture the global response generated by the blast load. This is because the deformation and shattering of the facade dissipates much of the energy from the blast.

Fu [35] and Jeyarajan et al. [33] converted blast pressure into line loads and applied them to the frame beam elements. Fu [8] created a Visual Basic program in order to do this for the entire frame. Jeyarajan et al. [33] simulated an explosion 20 m away from the building and assumed constant blast pressure over the height over the building. The pressure was then converted to line loads only for the front of the building.

Shi et al. [36] purposed a very different method in order to apply the blast load and take into account both partial damage and global response. Initial velocities are obtained for columns by assuming a deflection shape for the columns and solving a single degree of freedom (SDOF) dynamic system. For simplicity a deflection shape with plastic hinges at top, middle and bottom is assumed. Acceleration is assumed constant during the blast duration and by applying a blast impulse, velocities and displacement can be calculated. The time period of the blast is so small that the displacement is ignored and only the velocities is used as initial conditions in the global analysis. Damage is obtained from pressure impulse diagrams. These are analytically and/or numerically derived for each type of column. Using the maximum pressure and impulse from the blast loading a damage parameter from zero to one can be obtained from the pressure impulse diagram. Both initial velocities and damage have to be calculated separately for columns with different standoff distances. The damage is applied as reduces stiffness and yield strength in the end zones of the columns. The end zones are chosen based on the assumption that the columns will have shear failure.

In the study the method is applied to a small building with thee stories and two bays. For comparison a direct blast modeling by applying calculated reflected pressure directly to the column faces and an AP approach. The comparison shows that the purposed method shows a similar collapse response as the direct blast model, while the APM did not predict collapse. The method is highly efficient regarding computational cost compared to direct modeling of the blast. Is has more simplifications and for larger buildings it requires solving SDOF systems and P-I diagrams, as well as applying these results to all the columns in the global model.

Incident wave loading in Abaqus is intended to be used for modeling acoustic wave propagation through fluid elements that interact with structural elements, but it is possible to not model the fluid and apply the acoustic wave directly to solid, shell and beam elements. This will method will is explored and reported in chapter 5 and 6.

4.4.2 Modeling propagation of the blast wave

As the blast wave from an explosion propagates through air, it will be affected by reflections as it impinges on surfaces. This might increase the pressure as multiple waves are formed and joins, or the pressure might be decreased by shadowing surfaces. These effects can either be ignored, accounted for by factoring the blast pressure or directly model the propagation of the wave. Conwep does not take these effects into account.

Eulerian approach

In order to correctly model how the blast propagates through a building it is necessary to model the blast wave through the air using Eulerian elements. Conwep does not take into account shadowing or reflections [9].

In probably one of the most extensive blast progressive analyses done, Luccioni et al. [30] modeled an actual terrorist attack on a six story RC building. The RC frame and slabs was modeled using solid elements, the masonry facade with shell elements, and the air with Eulerian elements. A 400 kg TNT explosion was modeled just inside the building at the first floor. The analysis was conducted with the hydrocode AUTODYN The total number of elements was not reported. This approach clearly has a very high computational cost, but comparing the results of the analysis with photographs of the actual partial collapse of the building, it shows that it was able to reproduce the correct collapse of the building.

Shi et al. [38] compared reflected pressure off a concrete column using an Eulerian approach, to empirical values of TM 5M 5-855-1 [39]. It was found that for smaller columns (less than 1.6 m for rectangular and 3.0 m for circular) the empirical values overestimates the reflected pressure because of diffraction around the corner. Only 10 m standoff distance was tested. Al-Salloum et al. [40] compared the effects of modeling blast propagation though air on a local scale by modeling blast loading on one RC column. An Alternate Lagrangian Eulerian method was used for modeling a column of air surrounding the structural column. The RC column was modeled with solid elements and reinforcement with beam elements. The column was circular with about 0.5 m in diameter. The blast load was applied to the front face of the surrounding air column using Conwep loading in LS-DYNA. The same local model was also analyzed without modeling the air and applying the Conwep loading directly to the solid concrete elements. The study shows a negligible difference between the two methods. This points to that the difference in overestimation of reflected pressure from the empirical values found by Shi et al. [38] is negligible when comparing the structural response.

Confinement

If the explosion takes place inside the building the expansion of gases will build up a confined pressure in addition to the pressure of the shock front. This pressure load is much lower than the shock pressure but lasts longer. In a perfect confinement the gas pressure can be considered quasi-static. In real buildings the confinement is vented and the pressure will decay based on the amount of ventilation.

Al-Salloum et al. [40] included the effects of confinement by an internal explosion effect in the model by multiplying the Conwep blast pressure by a factor depending on an assumed level of venting. By assuming an opening of 36 m^2 in a 7056 m^3 floor an internal explosion factor of 1.7 was used No further verification or study of this assumption was made.

Adjacent buildings

No literature was found that takes into account reflections of surrounding buildings. If the adjacent buildings and the air in-between also were to be modeled in an Eulerian approach, this would further increase the already high computational cost. The blast pressures could be factorized by trying to take the surroundings into account. This could potentially be very uncertain, but could be necessary if large adjacent buildings are close.

4.4.3 Modeling the blast pressure

Characteristics of the blast wave, such as peak pressure, impulse and duration, may either be modeled directly by a thermodynamic equation of state or taken from empirical data based only on TNT-equivalent mass and stand-off distance.

In the extensive model of Luccioni et al. [30], the explosion was modeled in two stages. The explosion was modeled first with a 1D model using the Jones-Wilkins-Lee equation of state to generate a spherical blast wave. The blast wave was then mapped into 3D and the propagation of the blast wave was modeled using Eulerian air elements as described in above.

The most utilized method found, is implementing data from TM 5-855-1 [39] that is now replaced by UFC 4-010-02 DoD [12]. These data are based on analytical and experimental values and gives a variety of parameters for free air, air and surface explosions. Parameters may be read graphically from plots, and time-pressure curves can be fitted to these parameters as shown in. The same pressure curves can also be generated using commercial software like ATBLAST used by Fu [35].

4.4.4 Other physical effects from explosions

Explosions might cause more than just the shock wave and gas pressure mention earlier. It might propel fragments and create a ground shock wave. While these effects could do great damage on buildings and human life, they are less relevant when considering the structural response that could lead to progressive collapse. No search regarding these topics where conducted and no progressive collapse studies where observed taking these effects into account. However, there have been studies on progressive collapse caused by impact [1]. Electromagnetic Impulse (EMP) and radiation might also be caused by nuclear explosions, but this does not affect the structural response.

Weather conditions like atmospheric pressure, air flows and temperature will affect the pressure wave from an explosion. However these effects are only significant when the blast is so small or so far away from the building that they will likely not cause any significant structural damage [5].

One effect which is not related to the blast loading, but may be caused by explosions or other incidents, are fires. As seen in [2] fires may greatly reduce the capacity of structural members leading to collapse. Temperature effects was not researched in this study.

[1] ref WTC and nuclear cooling tower
[2] WTC

Chapter 5

Modeling

In order to study various analysis methods for progressive collapse and blast loading, FE models of a simple building where created.

5.1 Geometry

The building was a steel frame building with one-way reinforced concrete slabs. It consists of four by four bays, each 7.5 by 7.5 m. It had 5 stories, all three meters high with the same beam and column layout as shown in figure 5.1. As a simplification all columns were HFRHS300 and all beams HEB300. Section dimensions are shown in figure 5.2. Concrete slabs were 200 mm thick with 20 mm rebars with 120 mm spacing in both directions. The 20 mm were placed from the bottom of the slab. Slabs are the size of one bay and span in x-direction.

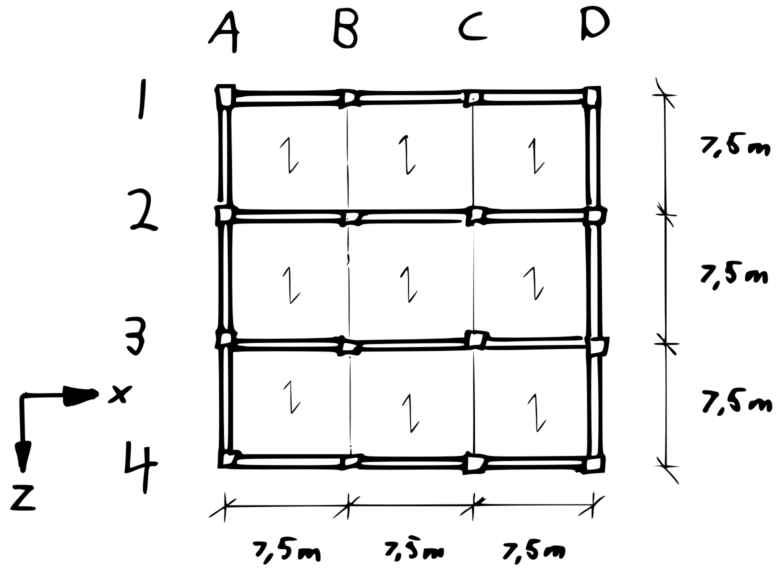
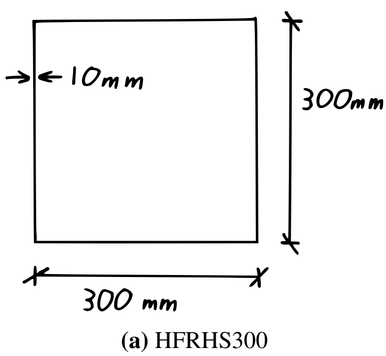
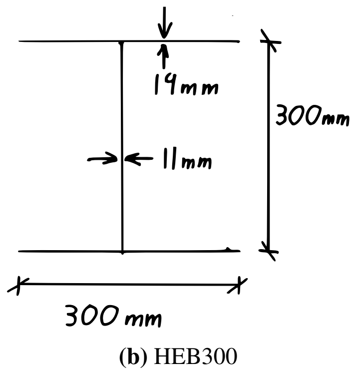


Figure 5.1: Plan view showing column and beam layout



(a) HFRHS300



(b) HEB300

Figure 5.2: Section dimensions

As a simplification all beam-column connections were modeled as fixed. This is not common for this type of building, but it simplifies the modeling as there is no need for lateral bracing. A more realistic approach would be to model the connections as pinned or with rotational stiffness and incorporate lateral bracing. The column bases are assumed fixed. Slabs are pinned along the beams.

5.2 FEM-model

Two different three-dimensional FE-models of the building was modeled. One model used beam elements to model the steel frame while the other used shell elements. They will be referred to as the 'beam model' and the 'shell model'. Only the steel frame and slabs were modeled. Both used shell elements for the slabs. Contact was not modeled. The analyses are only able to model the onset of collapse and not how it propagates as structural member make contact with each other or the ground.

5.2.1 Beam Model

Columns and beams were modeled with B31 elements. The B31 element is a three-dimensional linearly interpolated beam elements. It is a Timoshenko element, meaning it does include shear flexibility and stiffness [9].

The beams are connected to the columns with tie multi-point constraints. Both displacement DOFs and rotational DOFs are tied creating a fixed joint. The slabs are connected to the beams with tie connections only tying displacement DOFs.They are only tied on two opposite sided creating a one-way slab. The slabs where modeled in the same height as the beams without any offset. This is a simplification as the slabs should be resting on top of the beams. The slabs are assumed to be properly tied as to not slide off the beams. The slabs were modeled using the same element type as in the shell model.

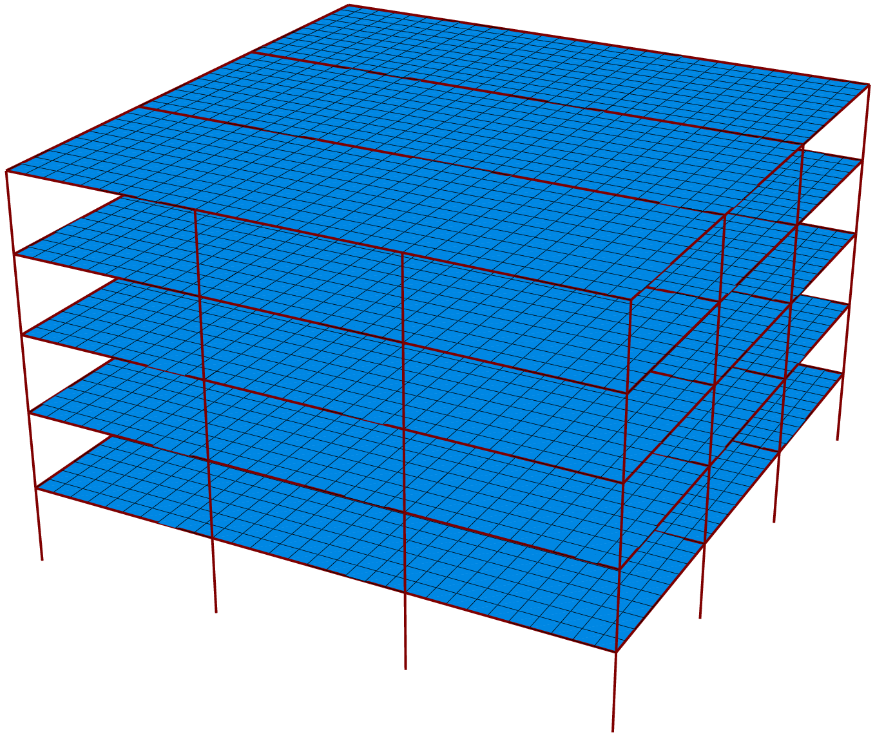


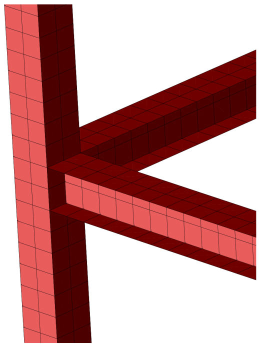
Figure 5.3: Beam model

5.2.2 Shell Model

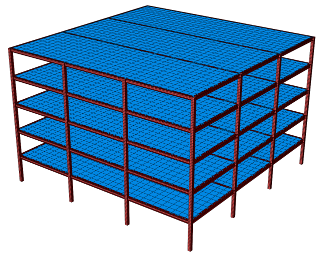
All members were modeled using a basic reduced integration shell element, S4R [9]. This is a four-node general-purpose shell element with hourglass control, finite membrane strains and reduced integration. Default numerical integration through the shell section, 5 point Simpson's rule, was used. General purpose means that the element may be used for both thin and thick shells. The element includes shear strains (Mindlin/Reissner theory) which is necessary for shells that are not thin [41]. The definition of a thin shell is that the thickness is less than about $1/15$ of the typical structural dimension. The slabs and column section are thin, but the beam is not.

The reduced integration reduces the computational cost, but gives rise to unphysical hourglass deformation modes. These modes allow deformation without strain energy in the element. For the S4R element these hourglass modes may propagate through the model giving large errors. Therefore artificial strain energy is introduced in order to control this.

The beam column connection is modeled by making sure that the beam and column mesh aligns and having them share nodes. This models a fixed joint. The slabs edges are pinned to the top center line on the beam sections using tie connections without tying rotational DOFs.



(a) Detail of shell frame model



(b)

Figure 5.4: Shell model

5.3 Materials

5.3.1 Steel

The steel was modeled with nonlinear isotropic hardening and damage. The yield strength was 355 MPa, Young's modules 210 000 MPa, Poisson's ratio 0.3, and density 7.8 $\mathrm{kg/m^3}$. Hardening parameters were K = 772 and n = 0.1733. An initial yield plateau until a strain of 0.024 was used before hardening started. The stress strain curve was created in Matlab and imported to Abaqus. Initiation of damage was also calculated using the Matlab script and equivalent plastic strain $\bar{\epsilon}^{pl}$ as a function of stress triaxiality was inputed to Abaqus. Linear damage based on effective plastic displacement with failure at 0.001 was used. [9]. As a simplification the rebar steel was taken to be the same as the frame steel. 0.05 mass proportional damping was used as in similar studies [35] [33].

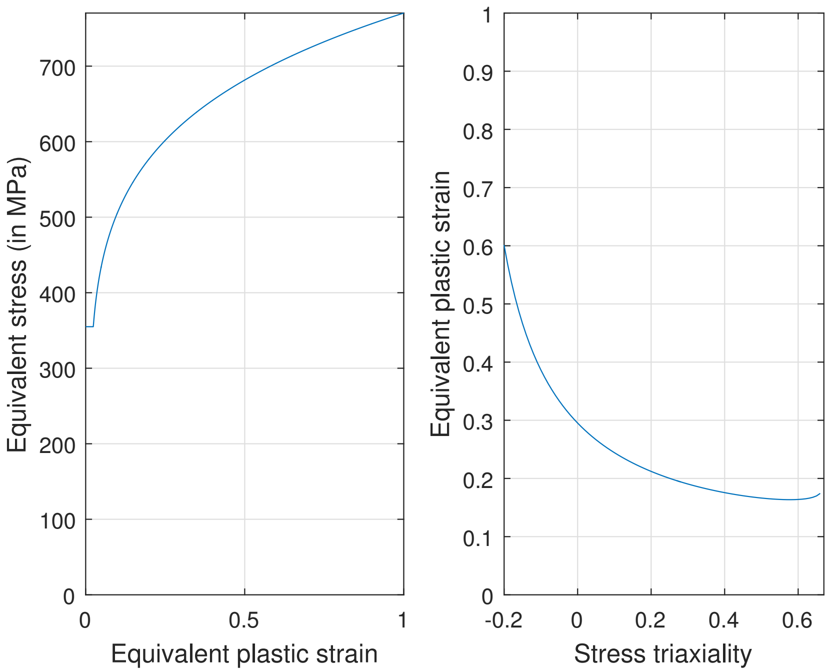


Figure 5.5: Steel model

5.3.2 Concrete

The concrete damaged plasticity model in Abaqus was first used, but this produced an error in several of the analyses. As the material modeling is not in focus of this thesis this was not explored further, but a very simple concrete model was used. The concrete was modeled as linear perfectly plastic with yield strength 30 MPa, Young's modules 35 000 MPa, Poisson's ratio 0.3, and density 2.5 $\mathrm{kg/m^3}$ No difference between tension and compression was made. The reinforcement was modeled as layers equivalent in the shell section. The same damping was applied as for the steel material.

5.4 Loading

The U.S. General Services Administration (GSA) recommends that the building is loaded with dead load (DL) and 25 % of the live load (LL) when conducting an AP analysis [37]. DL was taken as the weight of the materials and the LL was set to 2.0 $\mathrm{kN/m^2}$. No other loads, such as wind and snow, was included. All slabs was loaded with $0.25 * LL = 0.5\mathrm{kN/m^2}$ including the roof slabs.

5.4.1 Blast loading

An explosion equivalent of one ton TNT was modeled 10 m in x-direction from column D4 at ground level. The blast loading was modeled in two different ways, using an incident wave interaction and Conwep blast loading.

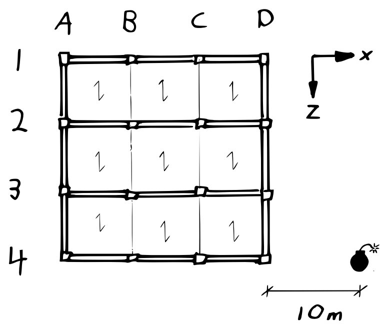


Figure 5.6: Plan view showing location of explosion

Conwep

Conwep (Conventional Weapons) is subtype of incident wave interaction in Abaqus. It models the blast wave of a spherical air blast or a hemispherical surface air blast without modeling the air. Hemispherical surface blast was used. The wave parameters are empirical data from TM 5-885-1 [39]. The only input parameters required for the interaction is the location of the blast source, the TNT-equivalent charge weight, and the surfaces that the blast acts on. It does not model shadowing effects or reflected waves. A limitation with Conwep is that it can only be applied to the surfaces of shells and solids, not beam elements. The Conwep model was therefore only used on the shell model.

Incident Wave Interaction

Even though Conwep is a type of incident wave interaction, the term incident wave interaction will be used excluding Conwep. It is used to simulate wave propagation through a fluid and interaction with structures. Since density of air is relatively low as a fluid it is possible to neglect the effect of the air and only model the structural components. The blast wave then propagates as a free air burst without any reflections or shadowing from surfaces.

Density of air is set to 1.225 $\mathrm{kg/m^3}$ and speed of sound to 340.29 $\mathrm{m/s}$. Since the fluid medium is not modeled, beam fluid inertia have to be specified for the beam sections in order to take drag into account. This is done by setting the density of the fluid ρ_f, drag coefficient of the section shape C_A and an effective radius of the section r. The added inertia of the section is then given by: $\pi r^2 \rho_f C_A$. The effective radius is set equal to the width of the beams and columns, 300 mm. A square section with rounded corners with $radius/width = 0.2$ has a theoretical drag coefficient of 1.2 for laminar flow [42]. This was used as an initial value. The incident wave load is not intended for use on open-section profiles, but was still used on the H-section beams.

The properties of the blast pressure is set by defining a source point and a reference point. A pressure amplitude is defined at the reference point. Parameters for the Friedlander equation (3.12) was taken from UFC 4-340-02 [12], and the b parameter was fitted in Matlab so that the impulse was correct. The Friedlander equation was then used to generate a pressure amplitude that was inputed to Abaqus. The propagation of the incident wave is spherical as of a free air burst. Since the explosion is at ground level, parameters for a surface burst is used. Since the propagation of a surface burst is hemispherical, this will be the same.

Blast pressure

A solid cube element, fixed at all nodes was applied Conwep loading in order to compare the Conwep blast pressure with Friedlander curve created. The single C3D8R cube solid element was placed with one side perpendicular to the blast source, and one of the other sides parallel. By using the output parameter IWCONWEP from Abaqus the pressure loading on a surface face is outputted. Reading this parameter on the perpendicular and parallel face of the element gives the maximum reflected pressure and the incident pressure. Figure 5.7 shows the incident pressure from the Conwep loading in Abaqus and the pressure curve the author has created from [12]. They show good correspondence.

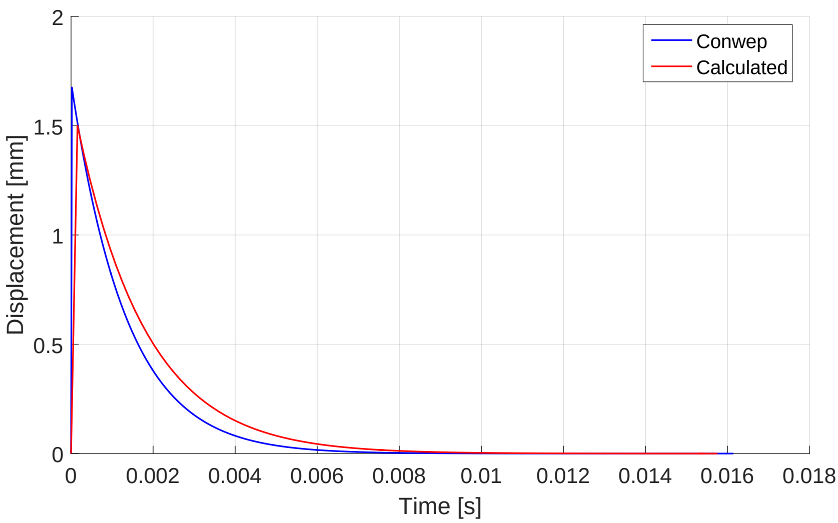


Figure 5.7: Incident pressure from Conwep loading and UFC [12]

5.5 Analyses

5.5.1 Alternate Path Analyses

Explicit time integration is often used in highly nonlinear problems because it is more stable than implicit integration. Analyses using explicit and implicit integration was compared to see if implicit integration may reduce computational cost. Nonlinear dynamic alternate path analyses where conducted using both implicit and explicit time integration, beam and shell models. First the DL and LL was applied static or quasi-static, then column D4 was removed and a dynamic step of two seconds to study the response. In Abaqus, implicit integration analysis was part of the Abaqus/Standard program, while Abaqus/Explicit was a separate program. Both are accessed within the same user interface, but their capabilities and limitations are different. Because of this, different modeling techniques have to be used for the implicit and explicit analyses.

Implicit Analysis

First a static step was conducted applying the DL and 0.25 LL. Then a column was deactivated using a model change interaction during a short dynamic step of 20 ms. During the model change the forces in the elements are linearly transfered to adjacent elements during the step. The column was removed by removing all but the top elements where the beams are connected in order not to remove the connection between the beams as purposed by GSA [37]. After that a two s dynamic step follows.

Explicit Analysis

The element removal interaction was not available in Abaqus/Explicit and manual removal of elements it not straight forward. The following approach was adopted: First a static analyses was done in Abaqus/Standard and moments and forces in the top of the column was extracted. A model in Abaqus/Explicit was created without the column but with the moments and forces extracted from the static analysis applied. A quasi-static step was ran applying the DL and LL. The step was ran for thee s with the load being applied with a smooth step amplitude over the first two s to avoid dynamic effects. Then the same procedure as the implicit analysis was done. Then the forces and moments applied from the static step was removed linearly over a period of 20 ms before a two s dynamic step.

5.5.2 Blast Analyses

Blast analyses were mainly done using explicit time integration. First a quasi-static analysis applying the DL and LL as in the AP analysis. Then the blast load was applied at the beginning of a dynamic step of two seconds.

5.5.3 Output

In order to be able to compare CPU times of implicit and explicit analyses, care had to be taken to make sure that they output about the same amount of data. Abaqus has two different output types, namely field and history output. Field output stores variables over a field for the entire model, or a select section of it. This data is typically used to produce spacial deformed or undeformed plots of the model with contour or symbol plots. History output stores data only for selected nodes or elements and are typically used to plot graphs or extracted for other operations.

Outputting field data for models with a large number of elements generates larges amounts of data. The amount of field data extracted was therefore limited to only deformations and status of element deletion.

For the implicit analysis the data was requested every increment. The explicit analysis uses a very large number of increments so the field output was requested at the same rate as the for the implicit analysis.

History was requested more frequent as it does not generate as much data as the field output. About 500 times during the analysis for the explicit analyses.

Chapter 6

Results

6.1 Alternate Path Models

The beam model was analyzed using the alternate path method, removing column D4. The difference between using implicit and explicit time integration was studied. The analyses was ran using different element sizes and a models with 10 stories in stead of 5 to see how this affected the results.

6.1.1 Response

Figure 6.1 shows vertical displacement above the removed column using explicit and implicit integration. The implicit curve is shifted to the right so the column removal happens at the same time as the end of the removal in the explicit analysis (t=3.02s). The response of the explicit is dampened out quicker but other than that they fit well. The explicit curve is more smooth because it has a higher frequency of history output, while the implicit outputs every increment. Figure 6.2 shows the same response, but zoomed in on the quasi-static and static loading step. There is a difference in deformation after loading because the forces added in place of the reaction forces in the removed column in the explicit model is not able to exactly balance applied load. But the quasi static solution is stable before the column removal showing that the quasi-static step is long enough to avoid dynamic effects. Similar results are observed for the 10 story model.

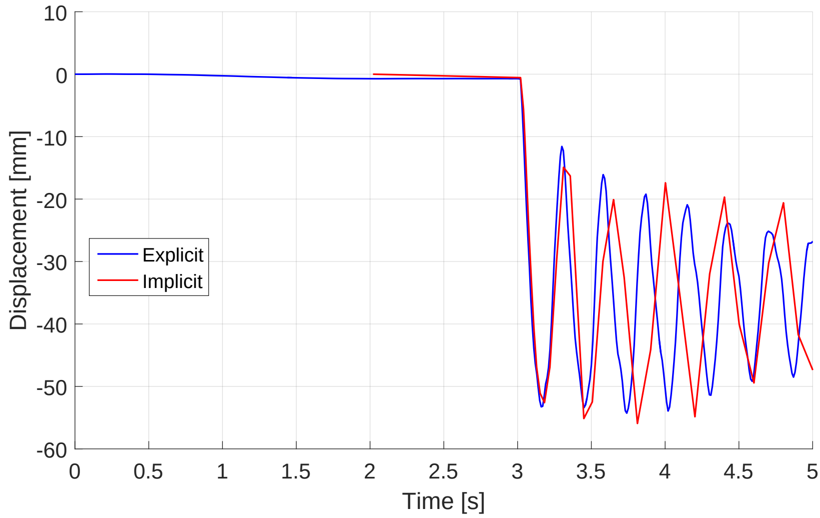


Figure 6.1: Vertical displacement above removed column in implicit and explicit beam model. Implicit curve is shifted so that column removal happens at the same time.

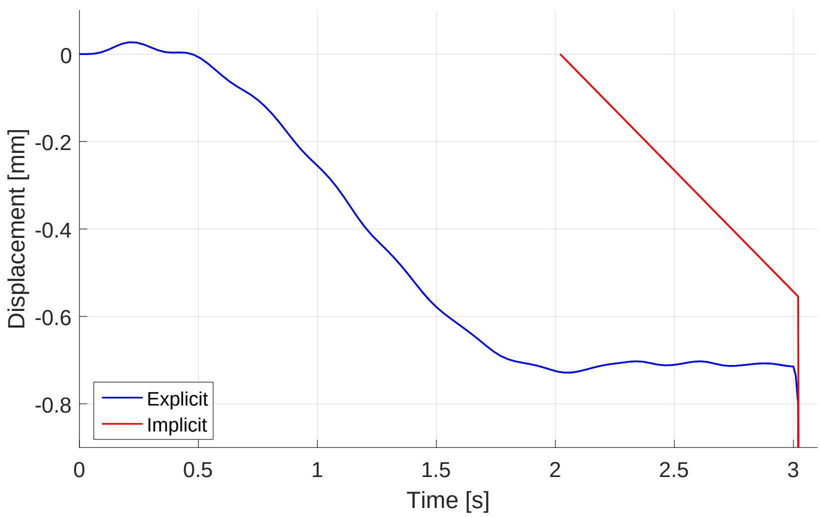


Figure 6.2: Vertical displacement above removed column in implicit and explicit beam model. Implicit curve is shifted so that column removal happens at the same time.

6.1.2 Seed

As shown in figure 6.3 there is not much difference in response with the tested element sizes with explicit integration. Both the frame and the slabs where meshed with the same seed. A 1 500 mm seed gives a slightly more flexible response but, not much. This corresponds to two beam elements for the columns and five for the beams. Similar small variations in response is observed with implicit integration.

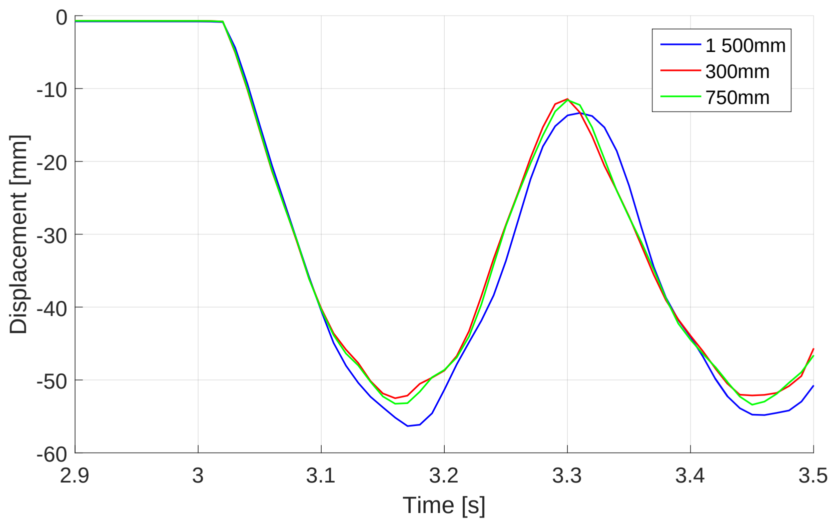


Figure 6.3: Vertical displacement of explicit analysis above removed column using different global seed

6.1.3 Ability to produce collapse

In order to verify that both the implicit and explicit analysis was able to reproduce collapse, the LL was increased linearly four seconds after column removal. Figure 6.4 shows the explicit model at t = 11.55 s. The collapse is initiated by plasticity and buckling of the interior columns followed by the exterior columns closest to the removed column. Since no contact and no ground was modeled the whole building starts falling indefinite until the analysis is stopped. Figure 6.5 shows vertical displacement, normalized total vertical reaction force, internal work and kinetic energy of the explicit analysis. The building does not collapse until about thee times the total load is applied after the column removal. The reason it is able to withstand so much is because the fixed joints are able to transfer a lot of force. The concrete is also overly stiff in tension because no cracking model is included. The buckling of the interior columns happens around 10 s and corresponds to the ‘bump’ that is observed in the kinetic energy at that time.

The response of the implicit and explicit model is very similar. The only difference is that the explicit dampens out vibrations faster, but the global response is the same as shown in 6.6.

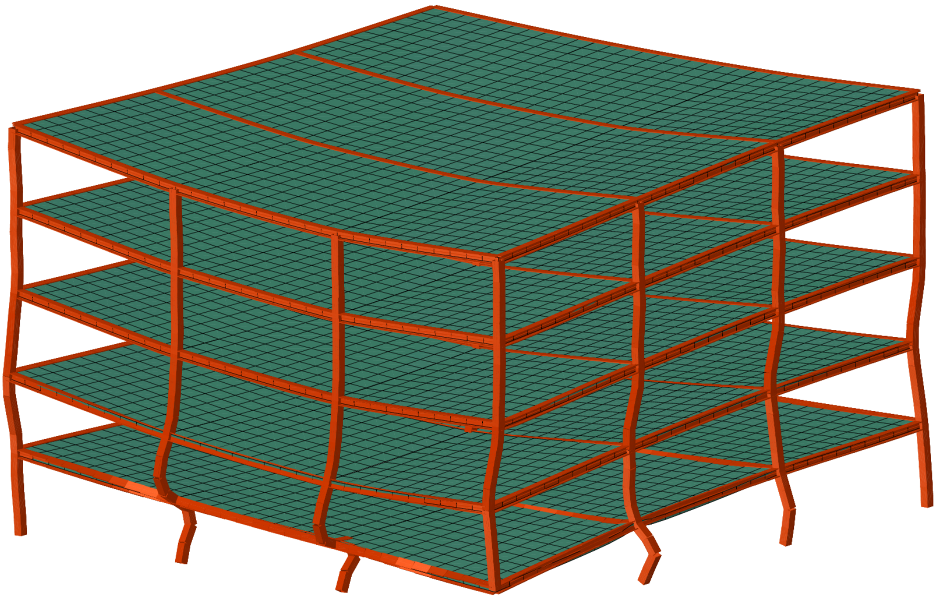


Figure 6.4: Collapse after removal of D4 and increasing the LL

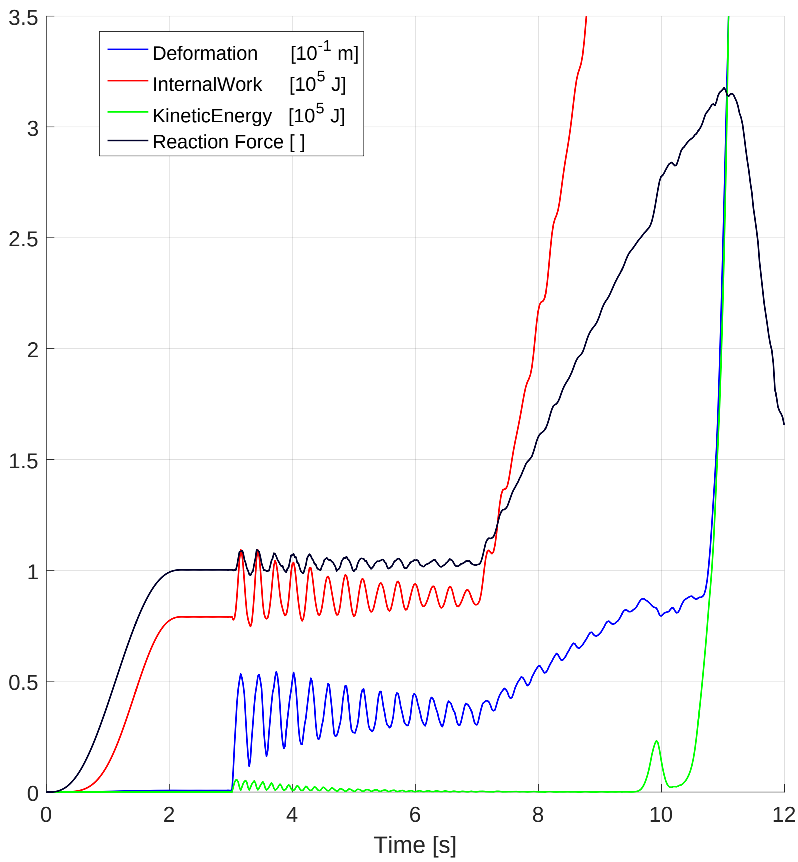


Figure 6.5: Vertical displacement, normalized total vertical reaction force, internal work and kinetic energy of collapse for the explicit analysis.

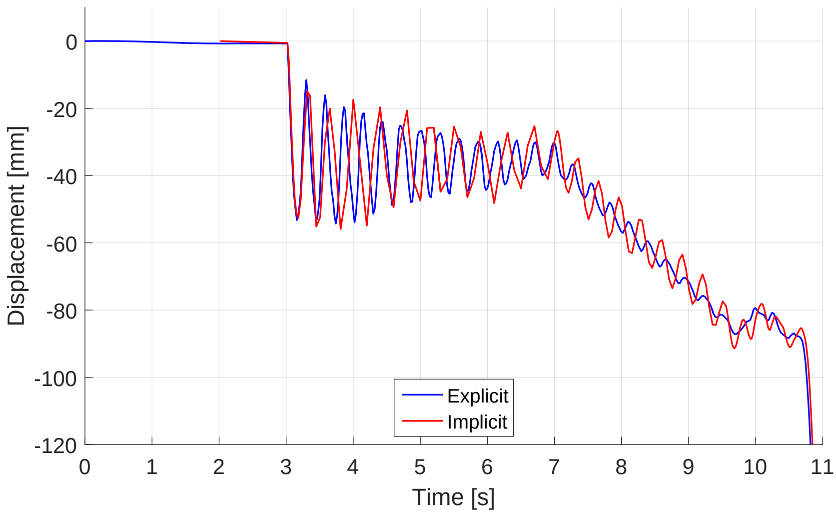


Figure 6.6: Vertical displacement at top of column removed in collapse analysis

6.1.4 Computational time

Computational cost is presented as CPU times normalized with respect to an explicit analysis with a 750 mm global seed. CPU times with varying seeds are presented in table 6.1. The explicit analysis is a little faster for the coarser mesh while the implicit is faster for the fines mesh with 300 mm elements. The reason for this is that the biggest influence on the explicit CPU time is the stable time increment which decreases with decreasing element size. If just increasing the size of the model while keeping the same element size the implicit CPU time increases more than the explicit. This is shown in table 6.2 by varying the building height. For the 15 story building the deformations had not stabilized within the prescribed time interval of the analyses. Running the analyses further reviled that the deformations did stabilize without any collapse, but it was close as all four interior columns had buckled in the ground floor. Since there was a lo of plasticity and buckling in the 15 story analysis the implicit analysis required a large number of increment in order to converge. This shows that the explicit analysis is often faster for nonlinear problems.

Table 6.1: Normalized CPU times with varying seed, 1 = 3 min

		Explicit integration		**Implicit integration**	
Seed [mm]	Elements [10^3]	CPU time	Stable increment [10^{-6} s]	CPU time	Increments
1 500	2	0.5	120	0.9	40
750	6	1	70	1.9	45
500	12	2	50	3.0	58
300	32	10	30	7.5	68

Table 6.2: Normalized CPU times, 1 = 3 min

		Explicit integration		**Implicit integration**	
Stories	Elements [10^3]	CPU time	Stable increment [10^{-6} s]	CPU time	Increments
5	6	1.0	70	1.9	45
10	11	1.4	70	4.2	58
15	17	9	70	35	202

For the collapse analyses the explicit was a lot faster. The implicit analysis was terminated when reaching 500 increments for the last loading step, 556 increments in total. The CPU time was then 1 h 45 min while the explicit finished in 8 min using about 70 000 increments. Because of the large deformations when the building starts to collapse a small time increment is needed and since every increment is costly because of the matrix factorization in the implicit integration, the analysis becomes very slow. When the implicit analysis was terminated 1.6 s after column removal, the explicit ran for two seconds. The last 300 implicit increments had only advanced the analysis 0.2 s.

6.1.5 Alternate Path analysis with the shell model

Implicit and explicit AP analyses was done on the shell model as well. As with the beam analyses, there was not a significant difference in the response between the explicit and implicit analyses. Figure 6.7 and 6.8 compares the response of the beam and shell model using explicit time integration. The response is similar but the beam model is stiffer. The shell initially has a larger displacement, but dampens out faster. The reason could be that since the shell model is modeling the beam and column section in 3D, this allows for local deformation of the column where the beam is joined. This will cause the joint to not be completely fixed as the beam element joints are. The total reaction force is similar. The reason for the difference in total vertical reaction force during the quasi-static loading is because of a difference in how the column is removed. In the beam model forces static forces are applied in stead of the column. In the shell model the column is there during quasi-static loading, and the fixation of the column support is removed instantaneously.

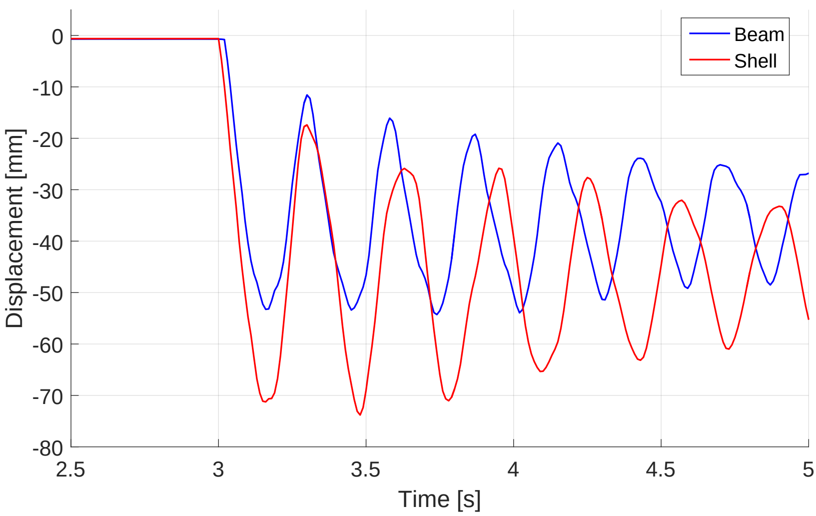


Figure 6.7: Vertical displacement at top of removed column for explicit shell model

Table 6.3: Normalized CPU times, 1 = 3 min

Stories	Elements [10^3]	Explicit integration CPU time	Explicit integration Stable increment [10^{-6} s]	Implicit integration CPU time	Implicit integration Increments
Beam	6	1.0	70	1.9	45
Shell	47	9	24	19	91

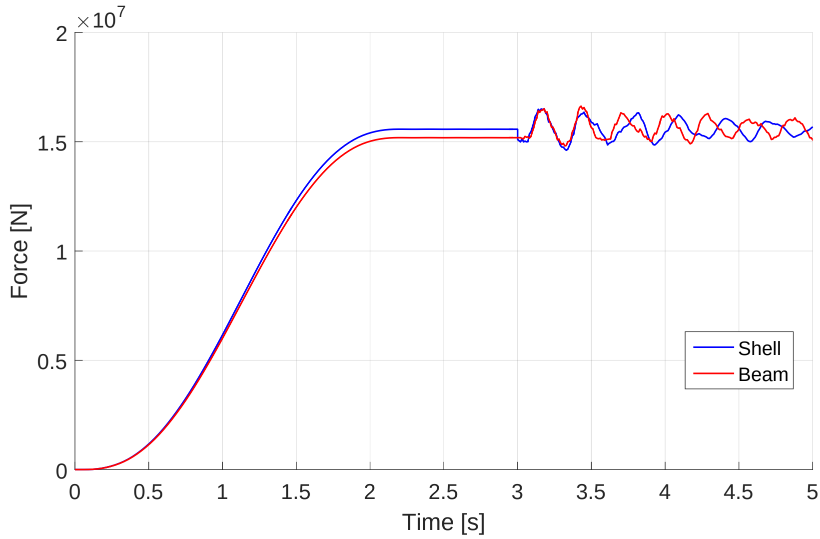


Figure 6.8: Total vertical reaction force for explicit shell model

6.2 Blast Loading on single Column

Before comparing the global beam and shell model, blast analyses on a single column was conducted. The column was the same as in the global building models, three meters high, 300 mm square steel section and fixed at both top and bottom. Both used the same element size, 150 mm. Verification of results against experimental results was not conducted.

6.2.1 Beam section parameters

Varying the drag coefficient and effective radius for the beam fluid inertia has a large impact on the response. 6.9 shows the lateral deflection of using drag coefficients of 0.9, 1.0 and 1.2. In later analysis 1.0 was used in the beam model as it was closer to the response of the shell model using Conwep loading. The other parameter affecting the blast loading on the beam elements is the effective radius of the section. The other section properties of the beam elements does not affect the incident wave load, only the response. This show that if a beam model is to be used to model blast with incident wave loading, it is important to calibrate the beam fluid inertia and effective radius against experiments or models that are assumed correct. Especially for sections more complex that circular or square.

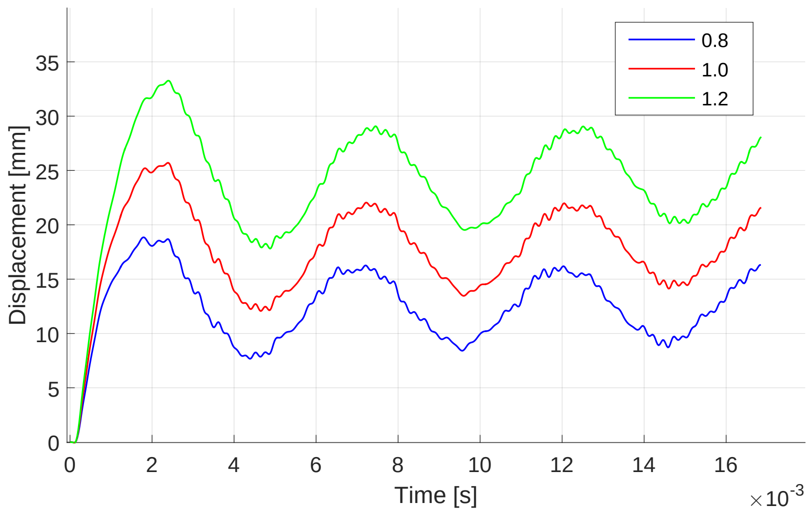


Figure 6.9: Lateral deformation at mid beam with varying drag coefficient

6.2.2 Comparing incident wave with Conwep loading

Beam incident wave vs shell Conwep

Figure 6.10 and 6.11 shows the response of the column for the beam model using incident wave loading and the shell model using Conwep loading. The deformation at the middle is not the same but similar. The reaction force is not similar. The response is visualized in figure 6.12. It shows how the deformation propagates as the blast hits the column from the bottom left corner. The beam model initially has a more local response only in the the lower part while the shell model starts deforming over the whole hight from the beginning. This causes a ‘whipping’ effect in the beam model that can be seen as a negative spike in the reaction force in figure 6.11. This also causes the beam model to have several hi-frequent deformation modes, while the shell model mostly oscillates in a global mode.

The cause of this difference in response is mainly because the incident wave and the Conwep wave travels with different velocity. The shock front of the incident wave loading travels at the speed of sound, while Conwep shock front travels at a varying supersonic speed determined from empirical data in the Conwep model. The correct velocity is that of the Conwep model. The velocity is always supersonic, and tends towards the speed of sound as the scaled distance Z gets increases. This means that this effect should become less as the distance increases of the charge weight decreases.

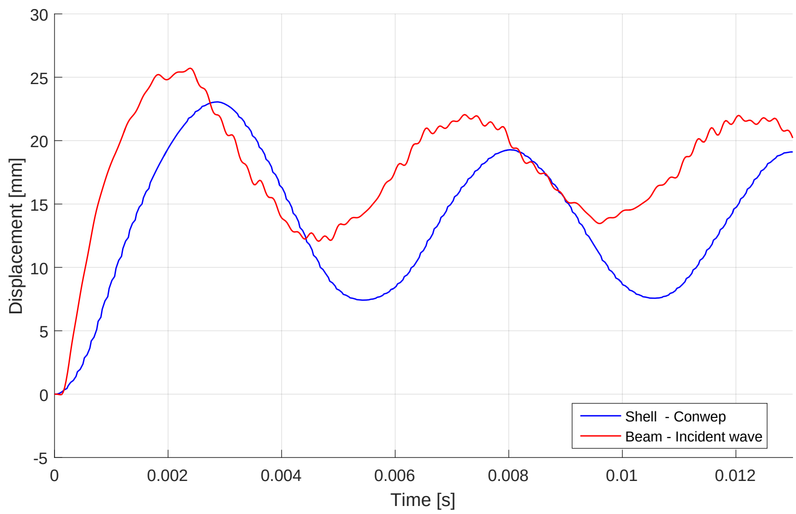


Figure 6.10: Horizontal deformation at middle of column.

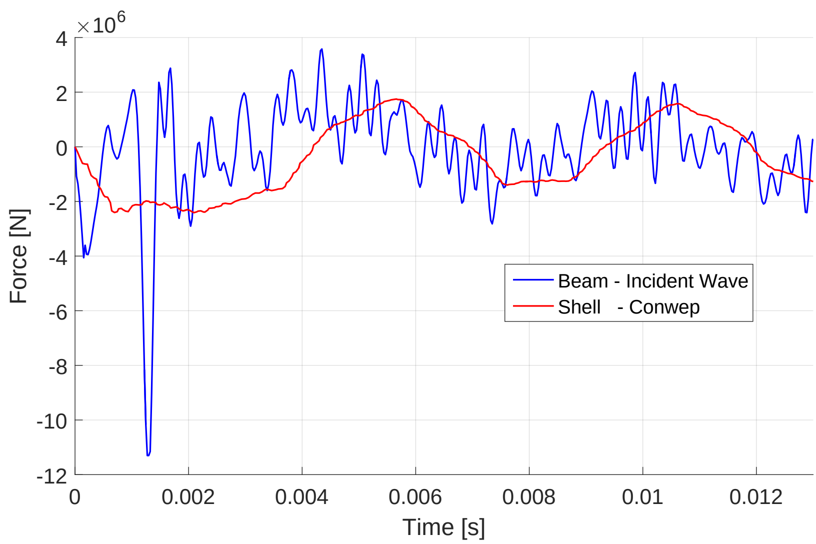
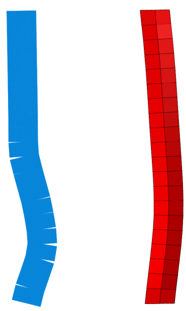
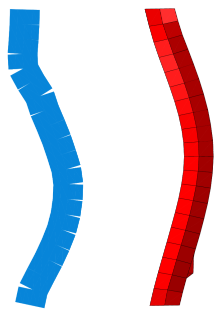


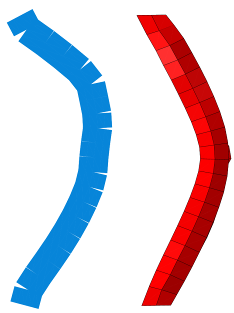
Figure 6.11: Total reaction force from top and base of column.



(a)



(b)



(c)

Figure 6.12: Deformation of column scaled by a factor of 30. Beam model is shown in blue and fig a), b) and c) is in chronoligical order. The blast comes directly from the left.

Incident wave load on shell elements

Figure 6.13 shows that by using incident wave loading the shell model gives substantially lower response. The incident wave blast load is not large enough to practice the column so the response oscillates around the initial configuration. The reason for this was found to be because the incident wave loading does not take into account the incident angle when it the blast wave impinges on a shell surface. The reflected pressure is supposed to have a maximum when the wave hits the surface perpendicular and the incident and reflected pressure should be equal when it hits parallel.

A simple test with analysis with shell elements perpendicular and parallel to the blast at three different distances was conducted. The reflected pressures is shown in figure 6.14. The pressure decreases as it hits a shell further away at a later time. The reflected pressure from the innocent wave loading does not depend on the angle of incident, but is always equal to the incident pressure. The time it takes for the incident blast wave to hit the shells is larger than for the Conwep blast wave and therefore the pressure is also lower for the same distance.

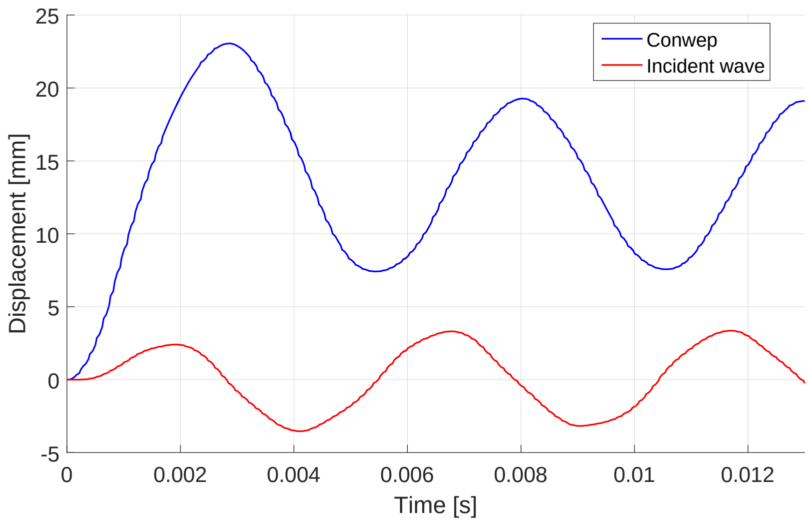


Figure 6.13: Horizontal displacement at center of column in shell model

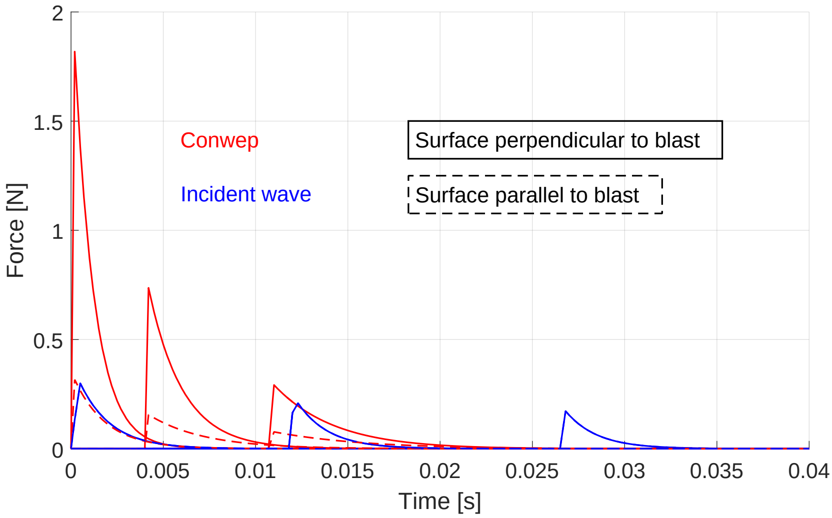


Figure 6.14: Horizontal displacement at center of column in shell model

6.2.3 Element size

Figure 6.15 and 6.16 shows the lateral deformation at the center of the column for different element sizes for both the beam and the shell model respectively.

For the beam model the response is somewhat smaller for a finer mesh and the high-frequency vibrations become less pronounced and becomes even more high-frequent. The is because each element is vibrating.

For the shell model the response becomes less stiff for a finer mesh. This is because more of the energy is taken up as local deformation of the section. Figure 6.17 shows how the section deforms different with different element sizes at the center of the column. If local buckling of the section is a possibility this deformation is very important to capture.

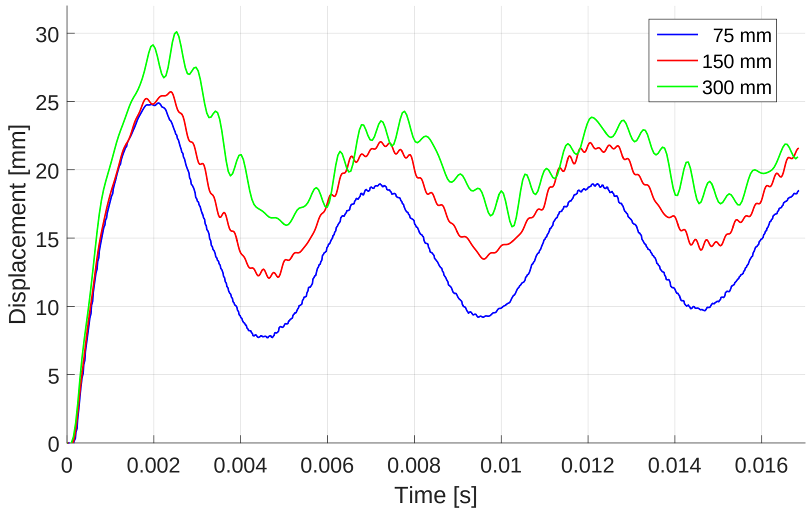


Figure 6.15: Displacement at middle of column with different element sizes for beam model.

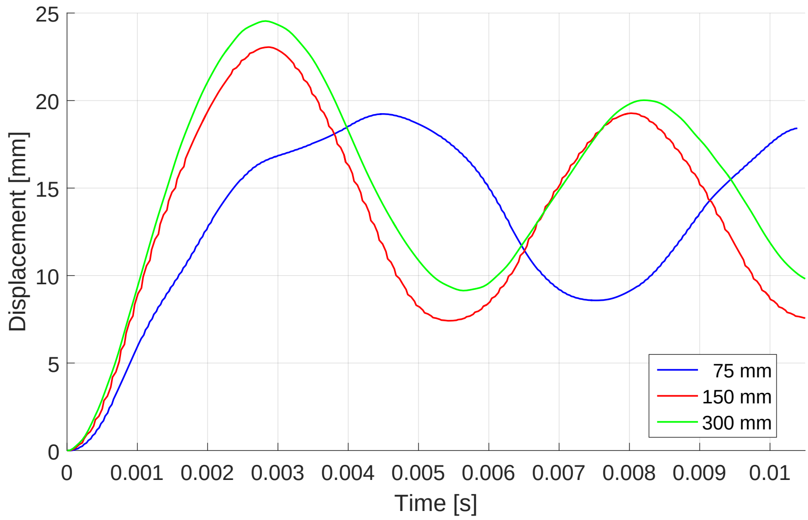
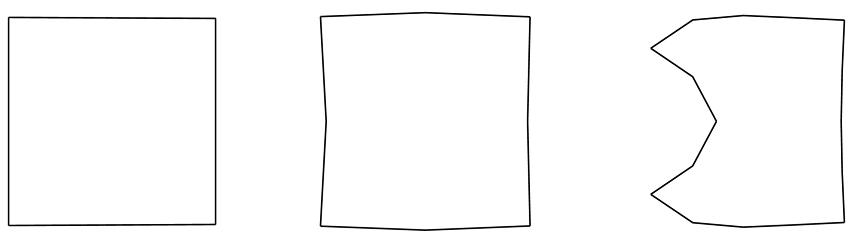


Figure 6.16: Displacement at middle of column with different element sizes for shell model.



(a) 300 mm **(b)** 150 mm **(c)** 75 mm

Figure 6.17: Deformation of column section at middle of column with different seed. Blast wave directly from left

6.2.4 Implicit time integration

Attempts where made to use implicit time integration to solve the beam model using implicit time integration in stead of explicit. The author was not successful at getting a convergent solution. For the very small part that did converge the time increment became so small (10^{-7}) that the method provides no benefit over explicit time integration.

6.3 Global blast models

Blast loading was applied to global building models. Only explicit integration was used and both beam and shell elements where tested for modeling the frame.

6.3.1 Response beam vs shell model

Figure 6.18, 6.19 and 6.20 shows some of the response of the blast model using beam or shell elements to model the frame. As seen the two models does not show the same response. Figure 6.19 shows that the global displacement response is larger for the shell model. Figure 6.18 shows that there is more energy in the beam model and 6.20. As seen in figure 6.19 the top of the beam model it displaced towards the blast side of the building, while the shell first is displaced away before it starts to swing back and forth. This is a similar response as seen for the single column model in sec 6.2 where the beam model deflects only locally first causing the top to come the other way, while the shell model the whole building deflects more together. In both models the sum of the reaction forces in the column bases become negative right after the blast meaning that the building is experiencing lift.

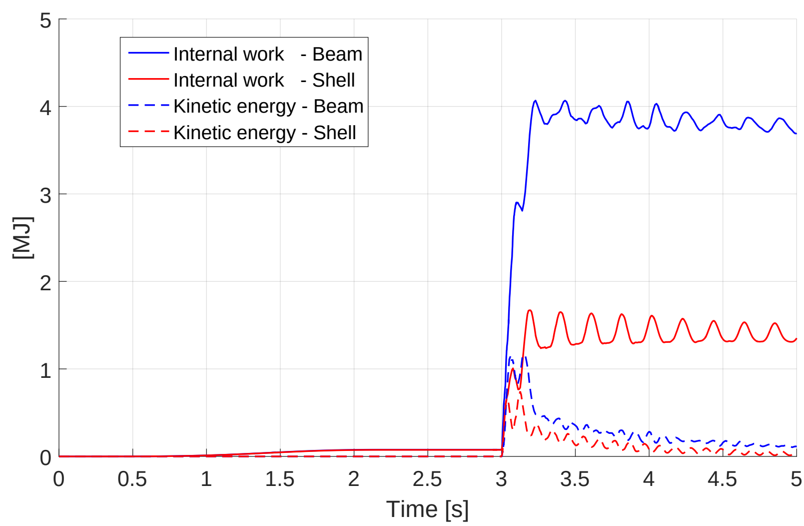


Figure 6.18: Internal work and kinetic energy of global beam and shell blast models

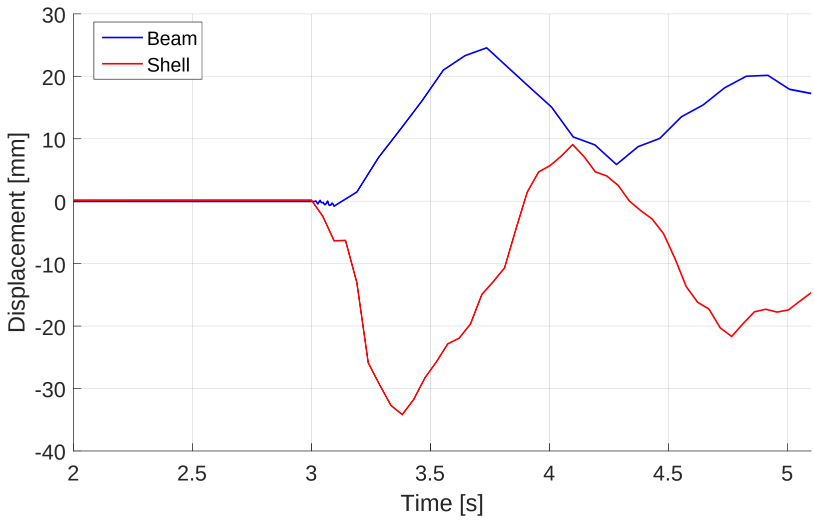


Figure 6.19: Vertical displacement in x direction at top of building in column D4

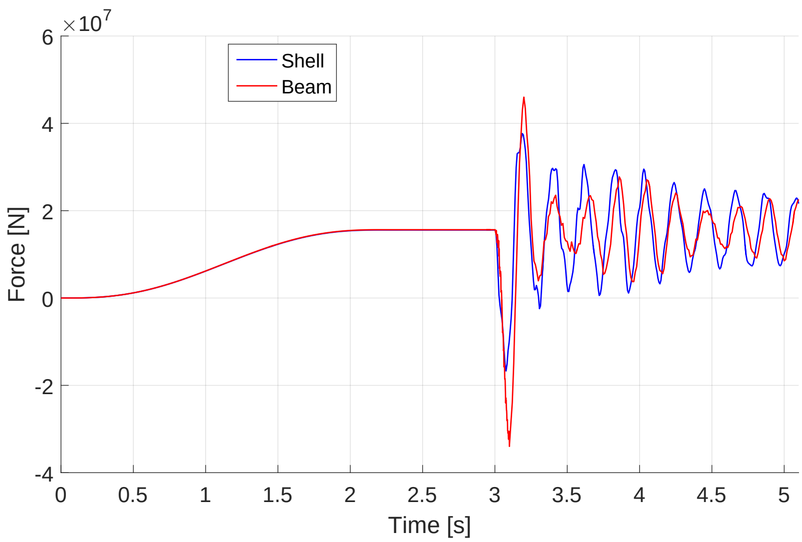
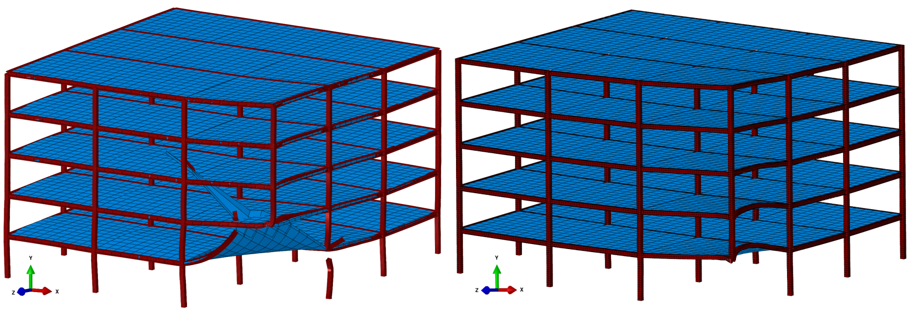


Figure 6.20: Total vertical reaction force

6.3.2 Large blast loading

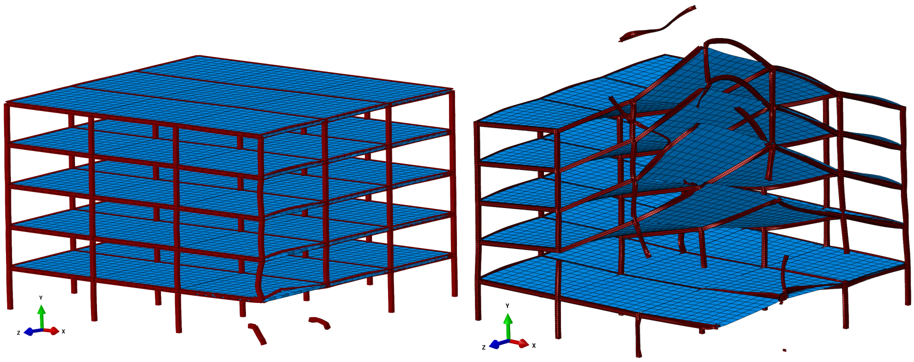
To see how the models behaved under large blast loading various standoff distances and charge weight where tested. The results clearly show that the beam and the shell model do not produce the same result. A one ton explosion at two meters from the building is shown in figure 6.21 with the beam model on the left and the blast model on the right. The shell model lost one column, but no other severe damage. The beam analysis terminated 0.5 s after the blast because of extreme deformations in the slab. This is because the concrete material is perfectly plastic and does not include any damage.

A 15 ton explosion 10 m from the building is shown in figure 6.22, again with the beam on the left and the shell model on the right. Here the beam model showed much less damage than the shell model. It lost two columns and where still standing. Large parts of the shell model was lifted several meters up as seen in the figure, and the building collapsed afterwards. Again the lack of damage in the concrete material makes this response rather unrealistic. If the concrete would have been damaged, the building would not have been lifted up that much.



(a) Beam model 50 ms after blast **(b)** Shell model two second after blast

Figure 6.21: Damage from one ton TNT with two m standoff distance



(a) Beam model two second after blast **(b)** Shell model 50 ms after blast

Figure 6.22: Damage from 15 ton TNT with 10 m standoff distance

6.3.3 CPU time

Table 6.4 shows normalized CPU time for blast analysis with both the beam and the shell model. The slab seed was kept constant at 750 mm for all models generating 100 elements per slab. This is because the response of the columns and beams are more interesting and if the same mesh size was used the finer models would have a very high number of elements because the are of the slabs is much larger than the frame. The longest CPU time was the shell model with seed 37.5 mm with over 30 hours.

Table 6.4: Normalized CPU times, 1 = 2 min

	Seed [mm]	Elements [10^3]	CPU time	Stable increment [10^{-6} s]
Beam model	750	6	1	70
	300	8	4	30
	150	10	9	15
	75	17	25	8
	37.5	29	80	4
Shell model	300	20	8	26
	150	43	15	23
	75	160	95	12
	37.5	626	885	5

Chapter 7

Conclusion and Further Work

7.1 Conclusion

The main approach to collapse analysis today is the alternate path method. A column is notionally removed in a structural analysis to see if the forces can be relocated to alternate paths. In order to correctly model the response of multi-story buildings, a global, three-dimensional, nonlinear, dynamic analysis have to be conducted. This may lead to large complex models, that have a high computational cost. One method so simplify the model and reduce computational cost is to use beam elements instead of shell or solid elements to model the frame of the building. It has been shown in the literature that this can predict an acceptable response. This fits with the results of this thesis.

Using implicit time integration instead of explicit have been studied to see if it could reduce computational cost. Modeling a steel frame building using both beam elements and shell elements did not show significant benefit of using implicit integration.

The alternate path method does not capture the structural response from an explosion or damage of multiple structural members. Other studies have concluded that this method is not always conservative for blast loading. Other studies have shown that it is possible to correctly model collapse of a building using large complex analyses with a very high computational cost. This thesis has tried to use the incident wave interaction in Abaqus in order to apply the correct blast load on beam elements. The method was compared with Conwep blast loading shell elements. Analyses was conducted on a moment stiff steel frame building with concrete slabs. The beam element model was unable to produce a satisfactory response compared with the shell element model using Conwep.

7.2 Further work

No experimental verification was conducted in this thesis. Verification against blast experiments on structural steel members may be conducted. This may be done in order to study what conditions are necessary for the incident wave loading to produce a correct response. Other blast loading approaches with beam member suggested in the literature may also be further studied or novel approaches may be adopted. Another thing that may be done is avoiding the quasi-static loading step. Computational cost may be saved by importing the results of a static analysis in stead of using a quasi-static step to apply the initial loads.

There are several topics not covered in this thesis relevant to blast and collapse analyses, and considerable simplifications were made in the analyses. Some suggestions about what may be included in a more realistic analysis are:

- *Modeling of joints.* They should be modeled as pinned or semi-rigid and realistic lateral bracing should be included in the model.

- *Better material models.* A proper concrete model is necessary and strain rate dependence may be included for the steel material.

Some suggestions on further topics to cover:

- *Fire.* An explosion might cause a fire that will weaken the materials.

- *Facade.* Many studies, including this thesis, does not model the facade of the building. This may affect the response.

- *The negative blast impulse.* This phase of the blast load was neglected in this study. This may be taken into account for steel frame buildings with flexible joints to see if it as an effect.

Bibliography

[1] ASCE. *Design for Physical Security: State of the Practice*. American Society of Civil Engineers, Reston, Virginia, 1999.

[2] Cynthia Pearson and Norbert Delatte. Ronan Point Apartment Tower Collapse and its Effect on Building Codes. *Journal of Performance of Constructed Facilities*, 19 (2):172–177, 2005.

[3] Ali Kazemi-Moghaddam and Mehrdad Sasani. Progressive collapse evaluation of Murrah Federal Building following sudden loss of column G20. *Engineering Structures*, 89:162–171, apr 2015.

[4] X Quan and N. K. Birnbaum. Computer Simulation of Impact and Collapse of New York World Trade Center North Tower on September 11. In *20th International Symposium on Ballistics*, pages 1–8, Orlando, Florida, 2002.

[5] DoD. *UFC 4-023-03: Design of Buildings To Resist Progressive Collapse*. U.S. Department Of Defence, 2009.

[6] Theodor Krauthammer. *Modern Protective Structures*. CRC Press, Taylor & Francis Group, Boca Raton, Florida, 2008.

[7] Per Kr Larsen. *Konstruksjonsteknikk - Laster og bæresystemer*. Tapir akademisk forlag, Trondheim, 2 edition, 2008.

[8] Feng Fu. *Advanced Modeling Techniques in Structural Design*. John Wiley & Sons, jun 2015.

[9] Dessault Systemès. ABAQUS 6.14 Documentation, 2014.

[10] Robert D. Cook, David S. Malkus, Michael E. Plesha, and Robert J. Witt. *Concepts and Applications of Finite Element Analysis*. Wiley, New York, 4 edition, 2002.

[11] Merriam-Webster.com. Explosion, 2016. URL `http://www.merriam-webster.com/dictionary/explosion`. Data accessed: 2016-06-01.

[12] DoD. *UFC 4-340-02: Structures to Resist the Effects of Accidental Explosions*. U.S. Department of Defence, 2008.

[13] Hrvoje Draganic and Vladimir Sigmund. Blast loading on structures. *Technical Gazzette*, 19(3):643–652, 2012.

[14] C. Cranz. *Lehrbruch der Ballistik*. Springer, Berlin, 1926.

[15] B Hopkinson. British Ordnance board minutes 13565, 1915.

[16] W. E. Baker. *Explosions in Air*. University of Texas press, Austin and London, 1973.

[17] Ru Lui, Buick Davison, and Andrew Tyas. A Study of Progressive Collapse in Multi-Storey Steel Frames. In *Structures Congress 20*, pages 1–9, New York, 2005. ASCE.

[18] HM Government. *The Building Regulations 2010. Structure. A3: Disproportionate Collapse*. HM Government, 2013.

[19] DoD. *UFC 4-010-01: Minimum Antiterrorism Standards for Buildings*. U.S. Department of Defence, 2013.

[20] Bruce R. Ellingwood, Robert Smilowitz, Donald O. Dusenberry, Dat Duthinh, H. S. Lew, and Nicholas J. Carino. *Best practices for reducing the potential for progressive collapse in buildings*. U.S. National Institute of Standards and Technology (NIST), 2007.

[21] Standard Norge. *NS-EN 1990:2002+NA2008 Eurocode: Basic of structural Design*. Standard Norge, Oslo, jan 2008.

[22] Standard Norge. *NS-EN 1991-1-7:2006+NA:2008 Eurocode 1: Actions of structures Part 1-7: General Actions, Accidental Actions*. Standard Norge, 2008.

[23] S M Marjanishvili. Progressive analysis procedure for progressive collapse. *Journal of Performance of Constructed Facilities*, 18(2):79–85, 2004.

[24] Kapil Khandelwal and Sherif El-Tawil. Pushdown resistance as a measure of robustness in progressive collapse analysis. *Engineering Structures*, 33(9):2653–2661, 2011.

[25] Alessandro Fascetti, Sashi K Kunnath, and Nicola Nisticò. Robustness evaluation of RC frame buildings to progressive collapse. *Engineering Structures*, 86:242–249, 2015.

[26] P E Graham Powell. Progressive collapse: Case studies using nonlinear analysis. In *Structures Congress 2005*, jan 2005.

[27] Edward L Wilson. *Three-Dimensional Static and Dynamic Analysis of Structures*. Computers and Structures, Inc., 3 edition, 2002.

[28] Yasser Alashker, Honghao Li, and Sherif El-Tawil. Approximations in Progressive Collapse Modeling. *Journal of Structural Engineering*, 137(9):914–924, 2011.

[29] Leslaw Kwasniewski. Nonlinear dynamic simulations of progressive collapse for a multistory building. *Engineering Structures*, 32(5):1223–1235, 2010.

[30] B M Luccioni, R D Ambrosini, and R F Danesi. Analysis of building collapse under blast loads. *Engineering Structures*, 26(1):63–71, 2004.

[31] Tarek H Almusallam, H M Elsanadedy, H Abbas, T Ngo, and P Mendis. Numerical Analysis for Progressice Collapse Potential of a Typical Framed Concrete Building. *International Journal of Civil & Environmental Engineering*, 10(02):36–42, 2010.

[32] Feng Fu. Progressive collapse analysis of high-rise building with 3-D finite element modeling method. *Journal of Constructional Steel Research*, 65(6):1269–1278, 2009.

[33] S Jeyarajan, J Y Richard Liew, and C G Koh. Analysis of Steel-Concrete Composite Buildings for Blast Induced Progressive Collapse. *International Journal of Protective Structures*, 6(3):457–485, 2015.

[34] H M Elsanadedy, T H Almusallam, Y R Alharbi, Y A Al-Salloum, and H Abbas. Progressive collapse potential of a typical steel building due to blast attacks. *Journal of Constructional Steel Research*, 101:143–157, 2014.

[35] Feng Fu. Dynamic response and robustness of tall buildings under blast loading. *Journal of Constructional Steel Research*, 80:299–307, 2013.

[36] Yanchao Shi, Li Zhong-Xian, and Hao Hong. A new method for progressive collapse analysis of RC frames under blast loading. *Engineering Structures*, 32(6):1691–1703, jun 2010.

[37] GSA. *Alternate Path Analysis & Design Guidelines for Progressive Collapse Resistance*. U.S. General Services Administration, 2013.

[38] Yanchao Shi, Hong Hao, and Zhong Xian Li. Numerical simulation of blast wave interaction with structure columns. *Shock Waves*, 17:113–133, 2007.

[39] Department of the Army. TM 5-855-1 Fundamentals of Protective Design for Conventional Weapons, 1998.

[40] Y A Al-Salloum, T H Almusallam, M Y Khawaji, T Ngo, H M Elsanadedy, and H Abbas. Progressive Collapse Analysis of RC Buildings Against Internal Blast. *Advances in Structural Engineering*, 18(12):2181–2192, 2015.

[41] Kolbein Bell. *An engineering approach to FINITE ELEMENT ANALYSIS of linear structural mechanics problems*. Fagbokforlaget, Bergen, Norway, 2014.

[42] Yunus Cengel and John Cimbala. *Fluid Mechanics Fundamentals and Applications*. Yunus Cengel, John Cimbala, New York, 3 edition, 2013.

Appendix

Selected Python Scripts

This appendix contains some of the scripts used in the thesis. *blastBeam.py* is an example of a script used to run explicit blast analyses with the beam model. The other two scripts *lib/func.py* and *lib/beam.py* are modules that must be placed in a folder called *lib* in the working directory of Abaqus. It imports an input file containing a steel material from a folder called *inputData* blastBeam.py then imports functions from the other two modules in *lib*. Module *lib/func.py* contains common functions used by all analysis. Module *lib/beam.py* contains functions related to the beam model. Similar modules was created for the shell model, and the single column models. All scripts used in this thesis are available online at `https://github.com/fsdalen/ProgressiveCollapse`.

blastBeam.py

```python
#Abaqus modules
from abaqus import *
from abaqusConstants import *


#========================================================#
#========================================================#
#                       CONTROLS                         #
#========================================================#
#========================================================#


modelName       = 'blastBeam'
cpus            = 8             #Number of CPU's

run             = 0

parameter       = 0
runPara         = 0



#=========== Geometry  ============#
#Size    4x4   x10(5)
x               = 4             #Nr of columns in x direction
z               = 4             #Nr of columns in z direction
```

```
y               = 5         #nr of stories


#=========== Step  ============#
quasiTime       = 3.0
blastTime       = 0.1
freeTime        = 2.0

qsSmoothFacor   = 0.75   #When smooth step reaches full amplitude during QS step

blastCol        = 'COLUMN_D4-1'
blastAmp        = 'blastAmp.txt'

precision    = SINGLE #SINGLE/ DOUBLE/ DOUBLE_CONSTRAINT_ONLY/ DOUBLE_PLUS_PACK
nodalOpt     = SINGLE #SINGLE or FULL


#=========== General  ============#
monitor         = 0         #Write status of job continusly in Abaqus CAE

#Live load
LL_kN_m         = -0.5      #kN/m^2 (-2.0)

#Mesh
seed            = 150.0     #Frame seed
slabSeed        = 750.0     #Slab seed
steelMatFile    = 'mat_7.5.inp' #Damage parameter is a function of element size

#Post
defScale        = 1.0
printFormat     = PNG       #TIFF, PS, EPS, PNG, SVG
animeFrameRate  = 5

qsIntervals     = 100
blastIntervals  = 100
freeIntervals   = 200

qsFieldIntervals    = 6
blastFieldIntervals = 22
freeFieldIntervals  = 22



#==========================================================#
#==========================================================#
#                      Perliminary                         #
#==========================================================#
#==========================================================#

import lib.func as func
import lib.beam as beam
reload(func)
reload(beam)

```

```
mdbName        = 'blastBeam'    #Name of .cae file


steel = 'DOMEX_S355'
concrete = 'Concrete'

#Set up model with materials
func.perliminary(monitor, modelName, steelMatFile)

M=mdb.models[modelName]




#==========================================================#
#==========================================================#
#                      Build model                         #
#==========================================================#
#==========================================================#

#Build geometry
beam.buildBeamMod(modelName, x, z, y, seed, slabSeed)



#=========== Quasi-static step  ============#

oldStep = 'Initial'
stepName = 'quasiStatic'
M.ExplicitDynamicsStep(name=stepName, previous=oldStep,
    timePeriod=quasiTime)


#Create smooth step for forces
M.SmoothStepAmplitude(name='smooth', timeSpan=STEP, data=(
    (0.0, 0.0), (qsSmoothFacor*quasiTime, 1.0)))

#Gravity
M.Gravity(comp2=-9800.0, createStepName=stepName,
    distributionType=UNIFORM, field='', name='Gravity',
    amplitude = 'smooth')

#LL
LL=LL_kN_m * 1.0e-3    #N/mm^2
func.addSlabLoad(M, x, z, y, stepName, LL, amplitude = 'smooth')



#=========== Blast step  ============#
#Create step
oldStep = stepName
stepName = 'blast'
M.ExplicitDynamicsStep(name=stepName, previous=oldStep,
    timePeriod=blastTime)

#Join surfaces to create blastSurf
```

```
lst = []
for inst in M.rootAssembly.instances.keys():
    if inst.startswith('BEAM') or inst.startswith('COLUMN'):
        lst.append(M.rootAssembly.instances[inst].surfaces['surf'])
    if inst.startswith('SLAB'):
        lst.append(M.rootAssembly.instances[inst].surfaces['botSurf'])
blastSurf = tuple(lst)
M.rootAssembly.SurfaceByBoolean(name='blastSurf', surfaces=blastSurf)

#Create blast
dic = {'A':0, 'B':1, 'C':2, 'D':3, 'E':4}
xBlast = dic[blastCol[7]]
zBlast = float(blastCol[8])-1
func.addIncidentWave(modelName, stepName,
    AmpFile= blastAmp,
    sourceCo =  (7500.0*xBlast + 10000.0, 0.0, 7500.0*zBlast),
    refCo =     (7500.0*xBlast +  1000.0, 0.0, 7500.0*zBlast))


#Remove smooth step from other loads
M.loads['Gravity'].setValuesInStep(stepName=stepName, amplitude=FREED)
func.changeSlabLoad(M, x, z, y, stepName, amplitude=FREED)


#=========== Free step  ============#
#Create step
oldStep = stepName
stepName = 'free'
M.ExplicitDynamicsStep(name=stepName, previous=oldStep,
    timePeriod=freeTime)


#====================================================#
#====================================================#
#                      Output                        #
#====================================================#
#====================================================#


#Detete default output
del M.fieldOutputRequests['F-Output-1']
del M.historyOutputRequests['H-Output-1']


#Displacement field output
M.FieldOutputRequest(name='U', createStepName='quasiStatic',
    variables=('U', ))

#Status field output
M.FieldOutputRequest(name='Status', createStepName='quasiStatic',
    variables=('STATUS', ))


#History output: energy
M.HistoryOutputRequest(name='Energy',
    createStepName='quasiStatic', variables=('ALLIE', 'ALLKE'))

```

```
#R2 at all col-bases
M.HistoryOutputRequest(createStepName='quasiStatic', name='R2',
    region=M.rootAssembly.sets['col-bases'], variables=('RF2', ))


#U2 at top of column closes to blast
M.HistoryOutputRequest(name=blastCol+'_top'+'U',
        createStepName='quasiStatic', variables=('U1','U2','U3'),
        region=M.rootAssembly.allInstances[blastCol].sets['col-top'],)


#Change frequency of output for all steps
func.changeHistoryOutputFreq(modelName,
    quasiStatic=qsIntervals, blast = blastIntervals, free=freeIntervals)
func.changeFieldOutputFreq(modelName,
    quasiStatic=qsFieldIntervals, blast = blastFieldIntervals,
    free=freeFieldIntervals)


#============================================================#
#============================================================#
#                    Save and run                            #
#============================================================#
#============================================================#
M.rootAssembly.regenerate()


#Create job
mdb.Job(model=modelName, name=modelName, numCpus=cpus, numDomains=cpus,
    explicitPrecision=precision, nodalOutputPrecision=nodalOpt)

#Run job
if run:
    #Save model
    mdb.saveAs(pathName = mdbName + '.cae')
    #Run model
    func.runJob(modelName)
    #Write CPU time to file
    func.readStaFile(modelName, 'results.txt')



#====================================================#
#====================================================#
#                    Post                            #
#====================================================#
#====================================================#

    print 'Post processing...'

    #Open ODB
    odb = func.open_odb(modelName)

    #Contour plots
    func.countourPrint(modelName, defScale, printFormat)

    #Animation
```

```
    func.animate(modelName, defScale, frameRate= animeFrameRate)

    #Energy plots
    func.xyEnergyPlot(modelName)

    #R2 at col base
    beam.xyColBaseR2(modelName,x,z)

    #U at top of col closes to blast
    beam.xyUtopCol(modelName, blastCol)

    print '   done'





#================================================================#
#================================================================#
#                    PARAMETER STUDY                             #
#================================================================#
#================================================================#
'''
Creates and runs mulpiple models and jobs varying a parameter.
'''

oldMod = modelName
if parameter:

    #=========== Seed  ============#
    paraLst = [1500, 500, 300]


    for para in paraLst:

        #New model
        modelName = 'beamBlastSeed'+str(para)

        mdb.Model(name=modelName, objectToCopy=mdb.models[oldMod])
        M = mdb.models[modelName]


        #=========== Change parameter  ============#
        beam.mesh(M, seed = para, slabSeedFactor=1.0)

        M.rootAssembly.regenerate()



        #=========== Create job and run  ============#

        #Create job
        mdb.Job(model=modelName, name=modelName,
            numCpus=cpus, numDomains=cpus,
            explicitPrecision=precision, nodalOutputPrecision=nodalOpt)


```

```
        if runPara:
            #Run job

            mdb.saveAs(pathName = mdbName + '.cae')
            func.runJob(modelName)
            func.readStaFile(modelName, 'results.txt')



            #=========== Post proccesing  ============#

            print 'Post processing...'

            #Energy
            func.xyEnergyPlot(modelName)

            #R2 at col base
            beam.xyColBaseR2(modelName,x,z)

            #U at top of col closes to blast
            beam.xyUtopCol(modelName, blastCol)






print '###########    END OF SCRIPT    ###########'
```

lib/func.py

```
# Abaqus modules
from abaqus import *
from abaqusConstants import *
from part import *
from material import *
from section import *
from optimization import *
from assembly import *
from step import *
from interaction import *
from load import *
from mesh import *
from job import *
from sketch import *
from visualization import *
from connectorBehavior import *
import odbAccess
import xyPlot
from jobMessage import ANY_JOB, ANY_MESSAGE_TYPE
import animation

#Python modules
import csv
from datetime import datetime
import glob






#============================================================#
#============================================================#
#                        PERLIMINARY                         #
#============================================================#
#============================================================#




def perliminary(monitor, modelName, steelMatFile='mat_75.inp'):
    #Makes mouse clicks into physical coordinates
    session.journalOptions.setValues(replayGeometry=COORDINATE,
        recoverGeometry=COORDINATE)

    #Print begin script to console
    print '\n'*6
    print '###########    NEW SCRIPT    ###########'
    print str(datetime.now())[:19]

    #Print status to console during analysis
    if monitor:
        printStatus(ON)

    #Create text file to write results in
```

```
    with open('results.txt', 'w') as f:
        None


    #=========== Set up model  ============#

    #Create model based on input material
    print '\n'*2
    mdb.ModelFromInputFile(name=modelName,
        inputFileName='inputData/'+steelMatFile)
    print '\n'*2

    #Deletes all other models
    delModels(modelName)

    #Close and delete old jobs and ODBs
    delJobs(exeption = steelMatFile)


    #=========== Material  ============#
    #Material names
    steel = 'DOMEX_S355'
    concrete = 'Concrete'


    M=mdb.models[modelName]
    createMaterials(M, mat1=steel, mat2=concrete)



#=========== Simple monitor  ============#
"""
simpleMonitor.py

Print status messages issued during an ABAQUS solver
analysis to the ABAQUS/CAE command line interface
"""
def simpleCB(jobName, messageType, data, userData):
    """
    This callback prints out all the
    members of the data objects
    """
    format = '%-18s  %-18s  %s'
    print '\n'*2
    print 'Message type: %s'%(messageType)
    members =  dir(data)
    for member in members:
        if member.startswith('__'): continue # ignore "magic" attrs
        memberValue = getattr(data, member)
        memberType = type(memberValue).__name__
        print format%(member, memberType, memberValue)

def printStatus(start=ON):
    """
    Switch message printing ON or OFF
    """

```

```
    if start:
        monitorManager.addMessageCallback(ANY_JOB,
            STATUS, simpleCB, None)
    else:
        monitorManager.removeMessageCallback(ANY_JOB,
            ANY_MESSAGE_TYPE, simpleCB, None)




#=========== Model ions  ============#

def delModels(modelName):
    """
    Deletes all models but modelName

    modelName= name of model to keep
    """
    if len(mdb.models.keys()) > 0:
        a = mdb.models.items()
    for i in range(len(a)):
        b = a[i]
        if b[0] != modelName:
            del mdb.models[b[0]]

def delJobs(exeption):
    """
    -Closes open odb files
    -Deletes jobs
    -Deletes .odb and .imp files
        (Because runnig Abaqus in Parallels often creates
        corrupted files)

    exeption = .inp file not to delete
    """
    #Close and delete odb files
    fls = glob.glob('*.odb')
    for i in fls:
        if len(session.odbs.keys())>0:
            session.odbs[i].close()
        os.remove(i)
    #Delete old input files
    inpt = glob.glob('*.inp')
    for i in inpt:
        if not i == exeption:
            os.remove(i)
    #Delete old jobs
    jbs = mdb.jobs.keys()
    if len(jbs)> 0:
        for i in jbs:
            del mdb.jobs[i]
    print 'Old jobs and ODBs have been closed.'
```

```python

#===========  Materials  ============#


def createMaterials(M, mat1, mat2):
    '''
    Adds damping to imported steel model
    Creates concrete and rebar steel

    M: model
    mat1, mat2, mat3: Name of materials
    '''

    damping = 0.05  #Mass proportional damping, same for all materials

    # Concrete
    mat2_Description = 'Elastic-perfect plastic'
    mat2_dens = 2.5e-09     #Density
    mat2_E = 35000.0        #E-module
    mat2_v = 0.3            #Poisson
    mat2_yield = 30.0       #Yield stress in compression





    #===========  Steel  ============#
    #Steel is already imported but needs damping
    M.materials[mat1].Damping(alpha=damping)

    #================ Concrete =================#
    M.Material(description=mat2_Description, name=mat2)
    M.materials[mat2].Density(table=((mat2_dens, ), ))
    M.materials[mat2].Elastic(table=((mat2_E, mat2_v), ))
    M.materials[mat2].Plastic(table=((mat2_yield, 0.0), ))
    M.materials[mat2].Damping(alpha=damping)








    #Concrete plasticity model, did not converge in static steps:(

    # mat2_yieldTension = 2.0 #Yield stress in compression
    # M.materials[mat2].ConcreteDamagedPlasticity(
    #     table=((30.0, 0.1, 1.16, 0.0, 0.0), ))
    #     #Dilatation angle, Eccentricity, fb0/fc0, K, Viscosity parameter
    # M.materials[mat2].concreteDamagedPlasticity.ConcreteCompressionHardening(
    #     table=((mat2_yield, 0.0), ))
    # M.materials[mat2].concreteDamagedPlasticity.ConcreteTensionStiffening(
    #     table=((mat2_yieldTension, 0.0), ))

```

```
#====================================================#
#====================================================#
#                     OTHER                          #
#====================================================#
#====================================================#



def setOutputIntervals(modelName,stepName, interval):
    '''
    Changes the number of output intervals for
    field and history output for a step
    '''
    M=mdb.models[modelName]

    for key in M.fieldOutputRequests.keys():
        M.fieldOutputRequests[key].setValuesInStep(
            stepName=stepName,
            numIntervals=interval)

    for key in M.historyOutputRequests.keys():
        M.historyOutputRequests[key].setValuesInStep(
            stepName=stepName,
            numIntervals=interval)


def changeHistoryOutputFreq(modelname, **kwargs):
    '''
    Changes the history output frequency for in all history outputs

    Input:
    modelName
    stepName=freq, stepName2=freq2 ...
    '''

    M=mdb.models[modelname]

    for step in kwargs:
        for hstOtpt in M.historyOutputRequests.keys():
            M.historyOutputRequests[hstOtpt].setValuesInStep(
                stepName=step, numIntervals=kwargs[step])

def changeFieldOutputFreq(modelname, **kwargs):
    '''
    Changes the field output frequency for all field outputs

    Input:
```

```
    modelName
    stepName=freq, stepName2=freq2...
    '''

    M=mdb.models[modelname]

    for step in kwargs:
        for fieldOtpt in M.fieldOutputRequests.keys():
            M.fieldOutputRequests[fieldOtpt].setValuesInStep(
                stepName=step, numIntervals=kwargs[step])






#=====================================================#
#=====================================================#
#                      LOADING                        #
#=====================================================#
#=====================================================#

#=========== Slab load ions for beam model ===========#


def addSlabLoad(M, x, z, y, step, load, amplitude=UNSET):
    '''
    Adds a surface traction to all slabs

    Parameters:
    M:         Model
    load:      Magnitude of load (positive y)
    x, z, y:   Nr of bays
    Step:      Which step to add the load
    Amplitude: default is UNSET
    '''

    #Create coordinate list
    alph = map(chr, range(65, 65+x)) #Start at 97 for lower case letters
    numb = map(str,range(1,z+1))
    etg = map(str,range(1,y+1))

    for a in range(len(alph)-1):
        for n in range(len(numb)-1):
            for e in range(len(etg)):
                inst = 'SLAB_'+ alph[a]+numb[n]+"-"+etg[e]
                M.SurfaceTraction(createStepName=step,
                    directionVector=((0.0, 0.0, 0.0), (0.0, 1.0, 0.0)),
                    distributionType=UNIFORM, field='', follower=OFF,
                    localCsys=None, magnitude= load,
                    name=inst,
                    region=M.rootAssembly.instances[inst].surfaces['
                        topSurf'],
                    traction=GENERAL, amplitude = amplitude)


def changeSlabLoad(M, x, z, y, step, amplitude):
```

```
    '''
    Change

    Parameters:
    M:          Model
    load:       Magnitude of load (positive y)
    x, z, y:    Nr of bays
    Step:       Which step to add the load
    Amplitude:  default is UNSET
    '''

    #Create coordinate list
    alph = map(chr, range(65, 65+x)) #Start at 97 for lower case letters
    numb = map(str,range(1,z+1))
    etg = map(str,range(1,y+1))

    for a in range(len(alph)-1):
        for n in range(len(numb)-1):
            for e in range(len(etg)):
                inst = 'SLAB_'+ alph[a]+numb[n]+"-"+etg[e]
                M.loads[inst].setValuesInStep(stepName = step,
                    amplitude = amplitude)









#=========== Blast ions  ============#


def addIncidentWave(modelName, stepName, AmpFile, sourceCo, refCo):
    airDensity = 1.225e-12     #1.225 kg/m^3
    soundSpeed =340.29e3     # 340.29 m/s

    M=mdb.models[modelName]

    #Pressure amplitude from file blastAmp.csv
    firstRow=1
    table=[]
    with open('inputData/'+AmpFile, 'r') as f:
        reader = csv.reader(f, delimiter='\t')
        for row in reader:
            if firstRow:
                firstRow=0
            else:
                table.append((float(row[0]), float(row[1])))
                blastTime = float(row[0])
    tpl = tuple(table)
    M.TabularAmplitude(name='Blast', timeSpan=STEP,
        smooth=SOLVER_DEFAULT, data=(tpl))


    #Source Point
    feature = M.rootAssembly.ReferencePoint(point=sourceCo)
```

```
    ID = feature.id
    sourceRP = M.rootAssembly.referencePoints[ID]
    M.rootAssembly.Set(name='Source', referencePoints=(sourceRP,))

    #Standoff Point
    feature = M.rootAssembly.ReferencePoint(point=refCo)
    ID = feature.id
    standoffRP = M.rootAssembly.referencePoints[ID]
    M.rootAssembly.Set(name='Standoff', referencePoints=(standoffRP,))


    #Create interaction property
    M.IncidentWaveProperty(name='incidentWave',
        definition=SPHERICAL, fluidDensity=airDensity, soundSpeed=
            soundSpeed)


    #Create incident Wave Interaction
    M.IncidentWave(name='incidentWave', createStepName=stepName,
        sourcePoint=M.rootAssembly.sets['Source'],
        standoffPoint=M.rootAssembly.sets['Standoff'],
        surface=M.rootAssembly.surfaces['blastSurf'],
        definition=PRESSURE, interactionProperty='incidentWave',
        referenceMagnitude=1.0, amplitude='Blast')


    #Set model wave formulation (does not matter when fluid is not modeled
        )
    M.setValues(waveFormulation=TOTAL)



def addConWep(modelName, TNT, blastType, coordinates,timeOfBlast, stepName
    ):
    '''
    blastType = AIR_BLAST SURFACE_BLAST
    name of surf must be blastSurf

    timeoOfBlast, NB: total time
    TNT in tonns
    '''
    M=mdb.models[modelName]

    #Create interaction property
    M.IncidentWaveProperty(definition= blastType,
        massTNT=TNT,
        massFactor=1.0e3,
        lengthFactor=1.0e-3,
        pressureFactor=1.0e6,
        name='conWep',)

    #Source Point
    feature = M.rootAssembly.ReferencePoint(point=coordinates)
    ID = feature.id
    sourceRP = M.rootAssembly.referencePoints[ID]
    M.rootAssembly.Set(name='Source', referencePoints=(sourceRP,))

```

```
    #Create ineraction
    M.IncidentWave(createStepName=stepName, definition=CONWEP,
        detonationTime=timeOfBlast, interactionProperty='conWep',
        name='conWep',
        sourcePoint=M.rootAssembly.sets['Source'],
        surface=M.rootAssembly.surfaces['blastSurf'])

















#==================================================#
#==================================================#
#                       APM                        #
#==================================================#
#==================================================#


def historySectionForces(M, column, stepName):
    #Section forces and moments of top element in column to be deleted
    elmNr = M.rootAssembly.instances[column].elements[-1].label
    elm = M.rootAssembly.instances[column].elements[elmNr-1:elmNr]
    M.rootAssembly.Set(elements=elm, name='topColElm')

    M.HistoryOutputRequest(name='SectionForces', createStepName=stepName,
        variables=('SF1', 'SF2', 'SF3', 'SM1', 'SM2',
        'SM3'), region=M.rootAssembly.sets['topColElm'],)





def replaceForces(M, x, z, column, oldJob, oldStep, stepName, amplitude):
    '''
    Remove col-base BC or col-col constraint
    and add forces and moments from static analysis to top of colum
    M          = Model
    column     = column to be deleted in APM
    oldJob     = name of static job
    oldSte     = name of static step
    amplitude  = name of amplitude to add forces with
    '''
```

```
    #Delete col-base BC or col-col constraint
    if column[-1] == '1':
        #Delete single BC for all column bases
        del M.boundaryConditions['fixColBases']
        #Create one BC for each column
        alph = map(chr, range(65, 65+x)) #Start at 97 for lower case
            letters
        numb = map(str,range(1,z+1))
        for a in alph:
            for n in numb:
                colSet = 'COLUMN_' + a + n + "-" + "1.col-base"
                M.DisplacementBC(amplitude=UNSET, createStepName=
                    'Initial', distributionType=UNIFORM, fieldName='',
                        fixed=OFF,
                    localCsys=None, name=colSet, region=
                    M.rootAssembly.sets[colSet], u1=0.0, u2=0.0, u3=0.0
                    , ur1=0.0, ur2=0.0, ur3=0.0)
        #Delete one BC
        del M.boundaryConditions[column+'.col-base']
    else:
        topColNr = column[-1]
        botColNr = str(int(topColNr)-1)
        constName = 'Const_col_col_'+ column[-4:-1]+botColNr+'-'+topColNr
        del M.constraints[constName]

    #Open odb with static analysis
    odb = open_odb(oldJob)

    #Find correct historyOutput
    for key in odb.steps[oldStep].historyRegions.keys():
        if key.find('Element_'+column) > -1:
            histName = key

    #Create dictionary with forces
    dict = {}
    histOpt = odb.steps[oldStep].historyRegions[histName].historyOutputs
    variables = histOpt.keys()
    for var in variables:
        value = histOpt[var].data[-1][1]
        dict[var] = value

    #Where to add forces
    region = M.rootAssembly.instances[column].sets['col-top']

    #Create forces
    M.ConcentratedForce(name='Forces',
        createStepName=stepName, region=region, amplitude=amplitude,
        distributionType=UNIFORM, field='', localCsys=None,
        cf1=dict['SF3'], cf2=-dict['SF1'], cf3=dict['SF2'])

    #Create moments
    M.Moment(name='Moments', createStepName=stepName,
```

```
        region=region, distributionType=UNIFORM, field='', localCsys=None,
        amplitude=amplitude,
        cm1=dict['SM2'], cm2=-dict['SM3'], cm3=dict['SM1'])







def getElmOverLim(odbName, var, stepName, var_invariant, limit,
        elsetName=None):
    """
    Returns list with value and object for all elements over limit
    odbName       = name of odb to read from
    elsetName     = None, (may be set to limit what part of the model
                    to read)
    var           = 'PEEQ' or 'S'
    stepName      = Last step in odb
    var_invariant = 'mises' if var='S'
    limit         = var limit for what elements to return
    """
    elset = elemset = None
    region = "over the entire model"
    odb = open_odb(odbName)

    #Check to see if the element set exists in the assembly
    if elsetName:
        try:
            elemset = odb.rootAssembly.elementSets[elsetName]
            region = " in the element set : " + elsetName;
        except KeyError:
            print 'An assembly level elset named %s does' \
                'not exist in the output database %s' \
                % (elsetName, odbName)
            odb.close()
            exit(0)

    #Find values over limit
    step = odb.steps[stepName]
    result = []
    for frame in step.frames:
        allFields = frame.fieldOutputs
        if (allFields.has_key(var)):
            varSet = allFields[var]
            if elemset:
                varSet = varSet.getSubset(region=elemset)
            for varValue in varSet.values:
                if var_invariant:
                    if hasattr(varValue, var_invariant.lower()):
                        val = getattr(varValue, var_invariant.lower())
                    else:
                        raise ValueError('Field value does not have 
                            invariant %s' % (var_invariant,))
                else:
                    val = varValue.data
                if ( val >= limit):
                    result.append([val, varValue])
```

```
        else:
            raise ValueError('Field output does not have field %s' % (
                results_field,))
    return (result)



def delInstance(M, elmOverLim, stepName):
    '''
    Takes a list of elements and deletes the corresponding columns and
        beams.
    M          = model
    elmOverLim = list of elements
    stepname   = In what step to delete instances
    '''

    instOverLim = []
    #Create list of all instance names
    for i in range(len(elmOverLim)):
        instOverLim.append(elmOverLim[i][1].instance.name)

    #Create list with unique names
    inst = []
    for i in instOverLim:
        if i not in inst:
            inst.append(i)

    #Remove slabs so they are not deleted
    instFiltered=[]
    for i in inst[:]:
        if not i.startswith('SLAB'):
            instFiltered.append(i)

    #Merge set of instances to be deleted
    setList=[]
    for i in instFiltered:
        setList.append(M.rootAssembly.allInstances[i].sets['set'])

    setList = tuple(setList)
    if setList:
        M.rootAssembly.SetByBoolean(name='rmvSet', sets=setList)
    else:
        print 'No instances exceed criteria'

    #Remove instances
    M.ModelChange(activeInStep=False, createStepName=stepName,
        includeStrain=False, name='INST_REMOVAL', region=
        M.rootAssembly.sets['rmvSet'], regionType=GEOMETRY)
```

```
#=============================================================#
#=============================================================#
#                          JOB                                 #
#=============================================================#
#=============================================================#


class clockTimer(object):
    """
    Class for taking the wallclocktime of an analysis.
    Uses the python ion datetime to calculate the elapsed time.
    """
    def __init__(self):
        self.model = None

    def start(self, model):
        '''
        Start a timer

        model = name of model to time
        '''
        self.startTime = datetime.now()
        self.model = model

    def end(self, fileName):
        '''
        End a timer and write result to file

        fileName = name of file to write result to
        '''
        t = datetime.now() - self.startTime
        time = str(t)[:-7]
        with open(fileName,'a') as f:
            text = '%s  wallClockTime:  %s\n' % (self.model, time)
            f.write(text)



def runJob(jobName):
    print 'Running %s...' %jobName

    '''
    Need to run jobs with an exeption in order to continue after riks step.
    The step is not completed but aborted when it reached max LPF.
    Also if maximum nr of increments is reach I still whant to be able to
    do post proccesing'''
```

```

    #Create and start timer
    timer = clockTimer()
    timer.start(jobName)

    #Run job
    try:
        mdb.jobs[jobName].submit(consistencyChecking=OFF)   #Run job
        mdb.jobs[jobName].waitForCompletion()
    except:
        print 'runJob Exeption:'
        print mdb.jobs[jobName].status

    #End timer and write result to file
    timer.end('results.txt')

    #=========== Display Job ============#
    #Open odb
    odb = open_odb(jobName)
    #View odb in viewport
    V=session.viewports['Viewport: 1']
    V.setValues(displayedObject=odb)
    # V.odbDisplay.display.setValues(plotState=(
    #   CONTOURS_ON_DEF, ))
    # V.odbDisplay.commonOptions.setValues(
    #   deformationScaling=UNIFORM, uniformScaleFactor=1)




def readMsgFile(jobName, fileName):
    '''
    Reads CPU time and nr of increments from .msg file
    and writes that to fileName

    jobName  = model to read CPU time for
    fileName = name of file to write result
    '''
    #Read .msg file
    with open(jobName+'.msg') as f:
        lines = f.readlines()

    #CPU time
    cpuTime = lines[-2]
    with open(fileName, 'a') as f:
        f.write(jobName + ' ' +cpuTime+'\n')

    #Nr of increments
    inc = lines[-22]
    with open(fileName, 'a') as f:
        f.write(jobName + ' ' +inc+'\n')



def readStaFile(jobName, fileName):
    '''
    Reads cpuTime and last stable time increment from .sta file.
```

```
    Prints result to fileName
    '''
    #Print CPU time to file
    with open(jobName+'.sta') as f:
        lines = f.readlines()

    cpuTime = lines[-7][32:40]
    stblInc = lines[-7][41:50]
    with open(fileName, 'a') as f:
        f.write(jobName + ' CPUtime ' +cpuTime+'\n')
        f.write(jobName + ' Stable Time Increment ' +stblInc+'\n')


#===================================================#
#===================================================#
#                       POST                        #
#===================================================#
#===================================================#


def open_odb(odbPath):
    """
    Enter odbPath (with or without extension)
    and get upgraded (if necesarly)

    Parameters
    odb = openOdb(odbPath)

    Returns
    open odb object
    """
    #Allow both .odb and without extention
    base, ext = os.path.splitext(odbPath)
    odbPath = base + '.odb'
    new_odbPath = None
    #Check if odb needs upgrade
    if isUpgradeRequiredForOdb(upgradeRequiredOdbPath=odbPath):
        print('odb %s needs upgrading' % (odbPath,))
        path,file_name = os.path.split(odbPath)
        file_name = base + "_upgraded.odb"
        new_odbPath = os.path.join(path,file_name)
        upgradeOdb(existingOdbPath=odbPath, upgradedOdbPath=new_odbPath)
```

```
        odbPath = new_odbPath
    odb = openOdb(path=odbPath, readOnly=True)
    return odb


def clearXY():
    '''
    Clears xy plots and data in session
    '''
    #Clear plots
    for plot in session.xyPlots.keys():
        del session.xyPlots[plot]

    #Clear xyData
    for data in session.xyDataObjects.keys():
        del session.xyDataObjects[data]


def XYplot(modelName, plotName, xHead, yHead, xyDat, reportFile='temp.txt'
    ):
    '''
    Saves xy data to a tab separated .txt file with headers

    modelName   = name of odbFile
    plotName    = name to give plot
    xHead       = x header
    yHead       = y header
    xyDat       = xy data to plot
    '''


    odb = open_odb(modelName)


    #=========== Report using Abaqus function  ============#
    session.writeXYReport(fileName=reportFile, appendMode=OFF, xyData=(
        xyDat, ))

    #=========== Fix report file  ============#
    #Create new better file than the strange Abaqus  output

    #Create fileName for output
    fileName = 'xyData_'+plotName+'_'+modelName+'.txt'

    #Read abaqus report file
    with open(reportFile, 'r') as f:
        lines = f.readlines()

    #Write formated data to new file
    a=None
    b=None
    with open(fileName, 'w') as f:
        f.write('%s\t%s\n' %(xHead, yHead))
        for line in lines:
            lst = line.lstrip().rstrip().split()
            if lst:
                try:
```

```
                    a = float(lst[0])
                    b = float(lst[1])
                except:
                    pass
                if type(a) and type(b) is float:
                    f.write(lst[0])
                    f.write('\t')
                    f.write(lst[1])
                    f.write('\n')
                    a=None
                    b=None


def fixAllTxtFilesInFolder():

    files = glob.glob('*.txt')

    for f in files:
        xHead      = 'Time [s]'
        yHead      = 'Displacement [mm]'
        dot        = f.find('.txt')
        name       = f[:dot]
        reportFile = f

        #Create fileName for output
        fileName = 'xyData_'+name+'.txt'

        #Read abaqus report file
        with open(reportFile, 'r') as f:
            lines = f.readlines()

        #Write formated data to new file
        a=None
        b=None
        with open(fileName, 'w') as f:
            f.write('%s\t%s\n' %(xHead, yHead))
            for line in lines:
                lst = line.lstrip().rstrip().split()
                if lst:
                    try:
                        a = float(lst[0])
                        b = float(lst[1])
                    except:
                        pass
                    if type(a) and type(b) is float:
                        f.write(lst[0])
                        f.write('\t')
                        f.write(lst[1])
                        f.write('\n')
                        a=None
                        b=None


def countourPrint(modelName, defScale, printFormat):
    '''
    Plots countour plots to file.

```

```
    modelName  =    name of odb
    defScale =  Deformation scale
    printFormat = TIFF, PS, EPS, PNG, SVG
    '''

    #Open odb
    odb = open_odb(modelName)
    #Create object for viewport
    V=session.viewports['Viewport: 1']
    #View odb in viewport
    V.setValues(displayedObject=odb)
    V.odbDisplay.display.setValues(plotState=(
        CONTOURS_ON_DEF, ))
    V.odbDisplay.commonOptions.setValues(
        deformationScaling=UNIFORM, uniformScaleFactor=defScale)

    #Print plots at the last frame in each step
    session.printOptions.setValues(vpBackground=OFF, compass=ON)
    for step in odb.steps.keys():
        V.odbDisplay.setFrame(step=step, frame=-1)
        #VonMises
        V.odbDisplay.setPrimaryVariable(
            variableLabel='S', outputPosition=INTEGRATION_POINT,
            refinement=(INVARIANT, 'Mises'), )
        session.printToFile(fileName='Cont_VonMises_'+step,
            format=printFormat, canvasObjects=(V, ))
        #PEEQ
        V.odbDisplay.setPrimaryVariable(
            variableLabel='PEEQ', outputPosition=INTEGRATION_POINT, )
        session.printToFile(fileName='Cont_PEEQ_'+step,
            format=printFormat, canvasObjects=(V, ))




def animate(modelName, defScale, frameRate):
    '''
    Animates the deformation with Von Mises contour plot
    Each field output frame is a frame in the animation
    (that means the animation time is not real time)

    modelName = name of job
    defScal = deformation scale
    frameRate = frame rate
    '''

    #Open odb
    odb = open_odb(modelName)
    #Create object for viewport
    V=session.viewports['Viewport: 1']

    #View odb in viewport
    V.setValues(displayedObject=odb)
    V.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))
    V.odbDisplay.commonOptions.setValues(
        deformationScaling=UNIFORM, uniformScaleFactor=defScale)
    V.odbDisplay.setPrimaryVariable(
```

```
        variableLabel='S', outputPosition=INTEGRATION_POINT,
        refinement=(INVARIANT, 'Mises'), )

    #Create and save animation
    session.animationController.setValues(animationType=TIME_HISTORY,
        viewports=(V.name,))
    session.animationController.play()
    session.imageAnimationOptions.setValues(frameRate = frameRate,
        compass = ON, vpBackground=ON)
    session.writeImageAnimation(fileName=modelName, format=QUICKTIME,
        canvasObjects=(V, )) #format = QUICKTIME or AVI

    #Stop animation
    session.animationController.stop()



def xyEnergyPlot(modelName):
    '''
    Prints External work, internal energy and kinetic energy for
    whole model

    modelName      = name of odb
    '''

    #Open ODB
    odb = open_odb(modelName)

    #Internal Work
    xyIW = xyPlot.XYDataFromHistory(odb=odb,
        outputVariableName='Internal energy: ALLIE for Whole Model',
        suppressQuery=True, name='xyIW')
    XYplot(modelName, plotName='InternalWork',
        xHead='Time [s]', yHead='Work [mJ]', xyDat=xyIW)

    #Kinetic Energy
    xyKE = xyPlot.XYDataFromHistory(odb=odb,
        outputVariableName='Kinetic energy: ALLKE for Whole Model',
        suppressQuery=True, name='xyKE')
    XYplot(modelName, plotName='KineticEnergy',
        xHead='Time [s]', yHead='Work [mJ]', xyDat=xyKE)
```

lib/beam.py

```python
# Abaqus modules
from abaqus import *
from abaqusConstants import *
from part import *
from material import *
from section import *
from optimization import *
from assembly import *
from step import *
from interaction import *
from load import *
from mesh import *
from job import *
from sketch import *
from visualization import *
from connectorBehavior import *
import odbAccess
import xyPlot
from jobMessage import ANY_JOB, ANY_MESSAGE_TYPE
import animation
import xyPlot

#Python modules
from datetime import datetime
import csv


import func




#===============================================================#
#===============================================================#
#                      Build beam model                         #
#===============================================================#
#===============================================================#


def buildBeamMod(modelName, x, z, y, seed, slabSeed):
    '''
    Builds a beam model without step
    '''

    col_height = 3000.0
    beam_len   = 7500.0

    steel = 'DOMEX_S355'
    concrete = 'Concrete'
    rebarSteel = steel

    M=mdb.models[modelName]


    #=========== Parts   ============#
```

```
    #Create Column
    createColumn(M, height=col_height, mat=steel, partName='COLUMN')

    #Create Beam
    createBeam(M, length=beam_len, mat=steel, partName='BEAM')

    #Create slab
    createSlab(M, t=200.0, mat=concrete, dim=beam_len,
        rebarMat=rebarSteel, partName='SLAB')

    #Add beam fluid inertia to beams and columns
    airDensity = 1.225e-12    #1.225 kg/m^3
    M.sections['HEB300'].setValues(useFluidInertia=ON,
        fluidMassDensity=airDensity, crossSectionRadius=300.0,
        lateralMassCoef=1.0)

    M.sections['HUP300x300'].setValues(useFluidInertia=ON,
        fluidMassDensity=airDensity, crossSectionRadius=300.0,
        lateralMassCoef=1.0)

    #=========== Sets and surfaces ============#
    #A lot of surfaces are created with the joints
    createSets(M, col_height)
    createSurfs(M)


    #=========== Assembly ============#
    createAssembly(M, x, z, y,
        x_d = beam_len, z_d = beam_len, y_d = col_height)


    #=========== Mesh ============#

    mesh(M, seed, slabSeed)


    #=========== Joints ============#
    createJoints(M, x, z, y,
        x_d = beam_len, z_d = beam_len, y_d = col_height)


    #=========== Fix column base ============#
    mergeColBase(M,x,z)
    M.DisplacementBC( createStepName='Initial',
        name='fixColBases', region= M.rootAssembly.sets['col-bases'],
        u1=0.0, u2=0.0, u3=0.0, ur1=0.0, ur2=0.0, ur3=0.0)






def createColumn(M, height, mat, partName):
    '''
    Creates a RHS 300x300 column

    M:        model
```

```
    height: height of column
    mat:     material
    '''

    sectName = "HUP300x300"

    #Create section and profile
    M.BoxProfile(a=300.0, b=300.0, name='Profile-1', t1=10.0, uniformThickness=ON)
    M.BeamSection(consistentMassMatrix=False, integration=
        DURING_ANALYSIS, material=mat, name=sectName, poissonRatio=0.3,
        profile='Profile-1', temperatureVar=LINEAR)

    #Create part
    M.ConstrainedSketch(name='__profile__', sheetSize=20.0)
    M.sketches['__profile__'].Line(point1=(0.0, 0.0), point2=(0.0, height))
    M.Part(dimensionality=THREE_D, name=partName, type=DEFORMABLE_BODY)
    M.parts[partName].BaseWire(sketch=M.sketches['__profile__'])
    del M.sketches['__profile__']

    #Assign section
    M.parts[partName].SectionAssignment(offset=0.0,
        offsetField='', offsetType=MIDDLE_SURFACE, region=Region(
        edges=M.parts[partName].edges.findAt(((0.0, 0.0,
        0.0), ), )), sectionName=sectName, thicknessAssignment=FROM_SECTION)

    #Assign beam orientation
    M.parts[partName].assignBeamSectionOrientation(method=
        N1_COSINES, n1=(0.0, 0.0, -1.0), region=Region(
        edges=M.parts[partName].edges.findAt(((0.0, 0.0, 0.0), ), )))




def createBeam(M, length, mat, partName):
    '''
    Creates a HEB 300 beam

    M:         model
    length:    lenght of beam
    mat:       material
    '''

    sectName = "HEB300"

    #Create Section and profile
    #HEB 550
    M.IProfile(b1=300.0, b2=300.0, h=300.0, l=150.0, name=
        'Profile-2', t1=19.0, t2=19.0, t3=11.0) #Now IPE profile, see ABAQUS for geometry definitions

    M.BeamSection(consistentMassMatrix=False, integration=
        DURING_ANALYSIS, material=mat, name=sectName, poissonRatio=0.3,
        profile='Profile-2', temperatureVar=LINEAR)

```

```
    #Create part
    M.ConstrainedSketch(name='__profile__', sheetSize=10000.0)
    M.sketches['__profile__'].Line(point1=(0.0, 0.0), point2=(length, 0.0)
        )
    M.Part(dimensionality=THREE_D, name=partName, type=DEFORMABLE_BODY)
    M.parts[partName].BaseWire(sketch=M.sketches['__profile__'])
    del M.sketches['__profile__']

    #Assign section
    M.parts[partName].SectionAssignment(offset=0.0,
        offsetField='', offsetType=MIDDLE_SURFACE, region=Region(
        edges=M.parts[partName].edges.findAt(((0.0, 0.0,
        0.0), ), )), sectionName=sectName, thicknessAssignment=
            FROM_SECTION)

    #Assign beam orientation
    M.parts[partName].assignBeamSectionOrientation(method=
        N1_COSINES, n1=(0.0, 0.0, -1.0), region=Region(
        edges=M.parts[partName].edges.findAt(((0.0, 0.0, 0.0), ), )))




def createSlab(M, t, mat, dim, rebarMat, partName):
    '''
    Creates a square slab with thickness 200.0

    M:          Model
    t:          Thickness of slab
    mat:        Material of section
    dim:        Dimention of square
    rebarMat:   Material of rebars
    '''

    sectName = "Slab"

    rebarDim = 20.0          #mm^2 diameter
    rebarArea = 3.1415*(rebarDim/2.0)**2        #mm^2
    rebarSpacing = 120.0         #mm
    rebarPosition = -80.0        #mm distance from center of section

    #Create Section
    M.HomogeneousShellSection(idealization=NO_IDEALIZATION,
        integrationRule=SIMPSON, material=mat, name=sectName, numIntPts=5,
        poissonDefinition=DEFAULT, preIntegrate=OFF,
        temperature=GRADIENT, thickness=t, thicknessField='',
        thicknessModulus=None, thicknessType=UNIFORM, useDensity=OFF)

    # Add rebars to section (both directions)
    M.sections[sectName].RebarLayers(layerTable=(
        LayerProperties(barArea=rebarArea, orientationAngle=0.0,
        barSpacing=rebarSpacing, layerPosition=rebarPosition,
        layerName='Layer 1', material=rebarMat),
        LayerProperties(barArea=rebarArea, orientationAngle=90.0,
        barSpacing=rebarSpacing, layerPosition=rebarPosition,
        layerName='Layer 2', material=rebarMat)),
        rebarSpacing=CONSTANT)
```

```
    #Create part
    M.ConstrainedSketch(name='__profile__', sheetSize= 10000.0)
    M.sketches['__profile__'].rectangle(point1=(0.0, 0.0),
        point2=(dim, dim))
    M.Part(dimensionality=THREE_D, name=partName, type=DEFORMABLE_BODY)
    M.parts[partName].BaseShell(sketch=M.sketches['__profile__'])
    del M.sketches['__profile__']

    #Assign section
    M.parts[partName].SectionAssignment(offset=0.0,
        offsetField='', offsetType=MIDDLE_SURFACE, region=Region(
        faces=M.parts[partName].faces.findAt(((0.0,
        0.0, 0.0), ), )), sectionName='Slab',
        thicknessAssignment=FROM_SECTION)

    #Assign Rebar Orientation
    M.parts[partName].assignRebarOrientation(
        additionalRotationType=ROTATION_NONE, axis=AXIS_1,
        fieldName='', localCsys=None, orientationType=GLOBAL,
        region=Region(faces=M.parts[partName].faces.findAt(
        ((0.1, 0.1, 0.0), (0.0, 0.0, 1.0)), )))




def createSets(M, col_height):
    '''
    Create part sets. Will be available in assembly as well.
    Naming in assembly: partName_an-e.setName (an-e are coordinates)

    M:        Model
    '''

    # Column base/top
    M.parts['COLUMN'].Set(name='col-base', vertices=
       M.parts['COLUMN'].vertices.findAt(((0.0, 0.0, 0.0),)))
    M.parts['COLUMN'].Set(name='col-top', vertices=
       M.parts['COLUMN'].vertices.findAt(((0.0, col_height, 0.0),)))

    #Column
    M.parts['COLUMN'].Set(edges=
       M.parts['COLUMN'].edges.findAt(((0.0, 1.0, 0.0), )),
        name='set')

    #Beam
    M.parts['BEAM'].Set(edges=
       M.parts['BEAM'].edges.findAt(((1.0, 0.0, 0.0), )),
        name='set')

    #Slab
    M.parts['SLAB'].Set(faces=
       M.parts['SLAB'].faces.findAt(((1.0, 1.0, 0.0), )),
        name='set')
```

```
def createSurfs(M):
    '''
    Create part surfaces. Will be available in assembly as well.
    Naming in assembly: partName_an-e.surfName (an-e are coordinates)

    Parameters
    M:       Model
    '''

    #Slab top and bottom
    M.parts['SLAB'].Surface(name='botSurf', side1Faces=
        M.parts['SLAB'].faces.findAt(((0.0, 0.0, 0.0), )))
    M.parts['SLAB'].Surface(name='topSurf', side2Faces=
        M.parts['SLAB'].faces.findAt(((0.0, 0.0, 0.0), )))


    #Circumferential beam surfaces
    circumEdges = M.parts['BEAM'].edges.findAt(((2000.0, 0.0, 0.0), ))
    M.parts['BEAM'].Surface(circumEdges=circumEdges, name='surf')


    #Create circumferential column surfaces
    circumEdges = M.parts['COLUMN'].edges.findAt(((0.0, 10.0, 0.0), ))
    M.parts['COLUMN'].Surface(circumEdges=circumEdges, name='surf')




def createAssembly(M, x, z, y, x_d, z_d, y_d):
    '''
    Creates an assembly of columns, beams and slabs.

    Parameters:
    M:       Model
    x,z,y:   Nr of bays and floors
    x_d:     Size of bays in x direction
    z_d:     Size of bays in z direction
    y_d:     Floor height
    '''

    #Create coordinate list
    #Letters go left to right (positive x)
    #Number top to bottom (positive z)
    alph = map(chr, range(65, 65+x)) #Start at 97 for lower case letters
    numb = map(str,range(1,z+1))
    etg = map(str,range(1,y+1))

    #Lists of all instances
    columnList = []
    beamList = []
    slabList = []

    #================ Columns =================#
    count=-1
    for a in alph:
        count = count + 1
        for n in numb:
```

```
            for e in etg:
                inst = 'COLUMN_' + a + n + "-" + e
                columnList.append(inst)
                #import and name instance
                M.rootAssembly.Instance(dependent=ON,
                    name= inst,
                    part=M.parts['COLUMN'])
                #Translate instance in x,y and z
                M.rootAssembly.translate(instanceList=(inst, ),
                    vector=(x_d*count , y_d*(int(e)-1),
                    z_d*(int(n)-1)))

    #================ Beams =================#
    #Beams in x (alpha) direction
    for a in range(len(alph)-1):
        for n in range(len(numb)-0):
            for e in range(len(etg)):
                inst = 'BEAM_'+ alph[a]+numb[n] + "-" + \
                    alph[a+1]+numb[n] + "-"+etg[e]
                beamList.append(inst)
                #import and name instance
                M.rootAssembly.Instance(dependent=ON,name=inst,
                    part=M.parts['BEAM'])
                M.rootAssembly.translate(instanceList=(inst, ),
                    vector=(x_d*a , y_d*(e+1), z_d*n))

    #Beams in z (numb) direction
    for a in [0,x-1]:
        for n in range(len(numb)-1):
            for e in range(len(etg)):
                inst = 'BEAM_'+ alph[a]+numb[n] + "-" + alph[a]+ \
                    numb[n+1] + "-"+etg[e]
                beamList.append(inst)
                # import and name instance
                M.rootAssembly.Instance(dependent=ON,name=inst,
                    part=M.parts['BEAM'])
                # Rotate instance
                M.rootAssembly.rotate(angle=-90.0, axisDirection=(
                    0.0,1.0, 0.0), axisPoint=(0.0, 0.0, 0.0),
                    instanceList=(inst, ))
                # Translate instance in x,y and z
                M.rootAssembly.translate(instanceList=(inst, ),
                    vector=(x_d*a , y_d*(e+1), z_d*n))


    #================ Slabs =================#
    for a in range(len(alph)-1):
        for n in range(len(numb)-1):
            for e in range(len(etg)):
                inst = 'SLAB_'+ alph[a]+numb[n] + "-"+etg[e]
                slabList.append(inst)
                M.rootAssembly.Instance(dependent=ON,name=inst,
                    part=M.parts['SLAB'])
                M.rootAssembly.rotate(angle=90.0,
                    axisDirection=(1.0,0.0, 0.0),
                    axisPoint=(0.0, 0.0, 0.0),
                    instanceList=(inst, ))
```

```
                    M.rootAssembly.translate(instanceList=(inst, ),
                        vector=(x_d*a,y_d*(e+1),z_d*(n)))



def mesh(M, seed, slabSeed):
    '''
    Meshes all parts
    Frame with seed and slabs with slabSeed
    '''

    #Same seed for beam and column
    seed1 = seed2 = seed
    seed3 = slabSeed

    analysisType = STANDARD  #Could be STANDARD or EXPLICIT
        #This only controls what elements are available to choose from

    element1 = B31 #B31 or B32 for linear or quadratic
    element2 = element1
    element3 = S4R  #S4R or S8R for linear or quadratic
                    #(S8R is not available for Explicit)

    #================= Column ==================#
    #Seed
    M.parts['COLUMN'].seedPart(minSizeFactor=0.1, size=seed1)
    #Change element type
    M.parts['COLUMN'].setElementType(elemTypes=(ElemType(
        elemCode=element1, elemLibrary=analysisType), ), regions=(
        M.parts['COLUMN'].edges.findAt((0.0, 0.0, 0.0), ), ))
    #Mesh
    M.parts['COLUMN'].generateMesh()

    #================= Beam ==================#
    #Seed
    M.parts['BEAM'].seedPart(minSizeFactor=0.1, size=seed2)
    #Change element type
    M.parts['BEAM'].setElementType(elemTypes=(ElemType(
        elemCode=element2, elemLibrary=analysisType), ), regions=(
        M.parts['BEAM'].edges.findAt((0.0, 0.0, 0.0), ), ))
    #Mesh
    M.parts['BEAM'].generateMesh()

    #================= Slab ==================#
    #Seed
    M.parts['SLAB'].seedPart(minSizeFactor=0.1, size=seed3)
    #Change element type
    M.parts['SLAB'].setElementType(elemTypes=(ElemType(
        elemCode=S4R, elemLibrary=analysisType, secondOrderAccuracy=OFF,
        hourglassControl=DEFAULT), ElemType(elemCode=S3R,
        elemLibrary=analysisType)),
        regions=(M.parts['SLAB'].faces.findAt((0.0, 0.0, 0.0), ), ))
    #Mesh
    M.parts['SLAB'].generateMesh()

    #Write nr of elements to results file
    M.rootAssembly.regenerate()
```

```
    nrElm = elmCounter(M)
    with open('results.txt','a') as f:
        f.write("%s Elements: %s \n" %(M.name, nrElm))




def elmCounter(M):
    '''
    Counts the total number of elements in model M.

    Returns:
    Number of elements
    '''
    nrElm = 0
    for inst in M.rootAssembly.instances.values():
        n = len(inst.elements)
        nrElm = nrElm + n
    return nrElm




def createJoints(M, x, z, y, x_d, z_d, y_d):
    '''
    Joins beams, columns and slabs with constraints.
    Beams are joined to columns with with MPC
    Columns are joined to columns with MPC
    Slabs are tied to beams with tie constrains.
    Slabs are only tied to beams in x direction to create one way slabs.

    Parameters:
    M:        Model
    x,z,y:    Nr of bays and floors
    x_d:      Size of bays in x direction
    z_d:      Size of bays in z direction
    y_d:      Floor height
    '''

    #MPC type for beam to column joints
    beamMPC = TIE_MPC    #May be TIE/BEAM/PIN (Tie will fix)
    colMPC = TIE_MPC



    #Set coordinates to Cartesian
    M.rootAssembly.DatumCsysByDefault(CARTESIAN)

    #Create coordinate list
    #Letters go left to right (positive x)
    #Number top to bottom (positive z)
    alph = map(chr, range(65, 65+x)) #Start at 97 for lower case letters
    numb = map(str,range(1,z+1))
    etg = map(str,range(1,y+1))

```

```
    #Lists of all instances
    columnList = []
    beamList = []
    slabList = []



    #=========== Beams to columns  ============#


    #Column to beam in x(alpha) direction
    for a in range(len(alph)-1):
        for n in range(len(numb)):
            for e in range(len(etg)):
                col = 'COLUMN_'+ alph[a]+numb[n] + "-" +etg[e]
                beam = 'BEAM_'+ alph[a]+numb[n] + "-" + \
                    alph[a+1]+numb[n] + "-"+etg[e]
                constrName = 'Const_col_beam_'+ alph[a]+numb[n] + "-" + \
                    alph[a+1]+numb[n] + "-"+etg[e]
                #MPC
                M.MultipointConstraint(controlPoint=Region(
                    vertices=M.rootAssembly.instances[col].vertices.findAt
                        (
                    ((a*x_d, (e+1)*y_d, n*z_d), ), )),\
                    csys=None, mpcType=beamMPC,
                    name=constrName, surface=Region(
                    vertices=M.rootAssembly.instances[beam].vertices.
                        findAt(
                    ((a*x_d, (e+1)*y_d, n*z_d), ), )),
                    userMode=DOF_MODE_MPC, userType=0)

    #Column to beam in negative x(alpha) direction
    for a in range(len(alph)-1, 0,-1):
        for n in range(len(numb)):
            for e in range(len(etg)):
                col = 'COLUMN_'+ alph[a]+numb[n] + "-" +etg[e]
                beam = 'BEAM_'+ alph[a-1]+numb[n] + "-" + \
                    alph[a]+numb[n] + "-"+etg[e]
                constrName = 'Const_col_beam_'+ alph[a]+numb[n] + "-" + \
                    alph[a-1]+numb[n] + "-"+etg[e]
                #MPC
                M.MultipointConstraint(controlPoint=Region(
                    vertices=M.rootAssembly.instances[col].vertices.findAt
                        (
                    ((a*x_d, (e+1)*y_d, n*z_d), ), )),
                    csys=None, mpcType=beamMPC,
                    name=constrName, surface=Region(
                    vertices=M.rootAssembly.instances[beam].vertices.
                        findAt(
                    ((a*x_d, (e+1)*y_d, n*z_d), ), )), userMode=
                        DOF_MODE_MPC, userType=0)

    #Column to beam in z(num) direction
    for a in [0,x-1]:
        for n in range(len(numb)-1):
            for e in range(len(etg)):
                col = 'COLUMN_'+ alph[a]+numb[n] + "-" +etg[e]
```

```
                beam = 'BEAM_'+ alph[a]+numb[n] + "-" + \
                    alph[a]+numb[n+1] + "-"+etg[e]
                constrName = 'Const_col_beam_'+ alph[a]+numb[n] + "-" + \
                    alph[a]+numb[n+1] + "-"+etg[e]
                #MPC
                M.MultipointConstraint(controlPoint=Region(
                    vertices=M.rootAssembly.instances[col].vertices.findAt(
                    ((a*x_d, (e+1)*y_d, n*z_d), ), )),
                    csys=None, mpcType=beamMPC, name=constrName,
                    surface=Region(
                    vertices=M.rootAssembly.instances[beam].vertices.findAt(
                    ((a*x_d, (e+1)*y_d, n*z_d), ), )),
                    userMode=DOF_MODE_MPC, userType=0)

    #Column to beam in negative z(num) direction
    for a in [0,x-1]:
        for n in range(len(numb)-1,0,-1):
            for e in range(len(etg)):
                col = 'COLUMN_'+ alph[a]+numb[n] + "-" +etg[e]
                beam = 'BEAM_'+ alph[a]+numb[n-1] + "-" +\
                    alph[a]+numb[n] + "-"+etg[e]
                constrName = 'Const_col_beam_'+ alph[a]+numb[n] + "-" + \
                    alph[a]+numb[n-1] + "-"+etg[e]
                #MPC
                M.MultipointConstraint(controlPoint=Region(
                    vertices=M.rootAssembly.instances[col].vertices.findAt(
                    ((a*x_d, (e+1)*y_d, n*z_d), ), )),\
                    csys=None, mpcType=beamMPC, name=constrName,
                    surface=Region(
                    vertices=M.rootAssembly.instances[beam].vertices.findAt(
                    ((a*x_d, (e+1)*y_d, n*z_d), ), )),
                    userMode=DOF_MODE_MPC, userType=0)




    #================ Column to column joints =============#

    for a in range(len(alph)):
        for n in range(len(numb)):
            for e in range(len(etg)-1):
                col = 'COLUMN_'+ alph[a]+numb[n] + "-" +etg[e]
                col2 = 'COLUMN_'+ alph[a]+numb[n] + "-" +etg[e+1]
                constrName = 'Const_col_col_'+ alph[a]+numb[n] + "-"+ \
                    etg[e] + "-"+etg[e+1]
                #MPC
                M.MultipointConstraint(controlPoint=Region(
                    vertices=M.rootAssembly.instances[col].vertices.findAt(
                    ((a*x_d, (e+1)*y_d, n*z_d), ), )),
                    csys=None, mpcType=colMPC, name=constrName,
                    surface=Region(
                    vertices=M.rootAssembly.instances[col2].vertices.
```

```
                        findAt(
                    ((a*x_d, (e+1)*y_d, n*z_d), ), )),
                    userMode=DOF_MODE_MPC, userType=0)




    #================ Slabs to beams ============#
    #Uses tie and not MPC


    #Join beam surfaces that are to be constrained
    for a in range(len(alph)-1):
        for n in range(len(numb)-1):
            for e in range(len(etg)):
                inst = 'SLAB_'+ alph[a]+numb[n] + "-"+etg[e]
                beam1 = 'BEAM_'+ alph[a]+numb[n] + "-" + \
                    alph[a+1]+numb[n] + "-"+etg[e]
                beam2 = 'BEAM_'+ alph[a]+numb[n+1] + "-" + \
                    alph[a+1]+numb[n+1] + "-"+etg[e]
                M.rootAssembly.SurfaceByBoolean(name=inst+'_beamEdges',
                    surfaces=(
                    M.rootAssembly.instances[beam1].surfaces['surf'],
                    M.rootAssembly.instances[beam2].surfaces['surf']
                    ))

    #=========== Slab edge surfaces  ============#
    for a in range(len(alph)-1):
        for n in range(len(numb)-1):
            for e in range(len(etg)):
                inst = 'SLAB_'+ alph[a]+numb[n] + "-"+etg[e]
                M.rootAssembly.Surface(name=inst+'_edges', side1Edges=
                    M.rootAssembly.instances[inst].edges.findAt(
                    ((x_d*a+1, y_d*(e+1), z_d*n), ),
                    ((x_d*a+1, y_d*(e+1), z_d*n+x_d), ),
                    ))

    #Tie slabs to beams (beams as master)
    for a in range(len(alph)-1):
        for n in range(len(numb)-1):
            for e in range(len(etg)):
                inst = 'SLAB_'+ alph[a]+numb[n] + "-"+etg[e]
                M.Tie(adjust=ON, master=
                    M.rootAssembly.surfaces[inst+'_beamEdges'],
                    name=inst, positionToleranceMethod=COMPUTED, slave=
                    M.rootAssembly.surfaces[inst+'_edges']
                    , thickness=OFF, tieRotations=OFF)



def mergeColBase(M,x,z):

    alph = map(chr, range(65, 65+x)) #Start at 97 for lower case letters
    numb = map(str,range(1,z+1))

    lst=[]
    for a in alph:
```

```
        for n in numb:
            inst = 'COLUMN_' + a + n + "-1"
            lst.append(M.rootAssembly.allInstances[inst].sets['col-base'])

    tpl = tuple(lst)
    M.rootAssembly.SetByBoolean(name='col-bases', sets=tpl)










#=====================================================#
#=====================================================#
#                     Output                          #
#=====================================================#
#=====================================================#


def xyColBaseR2(modelName,x,z):
    odb = func.open_odb(modelName)


    #=========== Get xy data for each colBot ============#
    alph = map(chr, range(65, 65+x)) #Start at 97 for lower case letters
    numb = map(str,range(1,z+1))
    count = 0
    lst=[]
    for a in alph:
        for n in numb:
            count = count + 1
            inst = 'COLUMN_' + a + n + "-1"
            name='Reaction force: RF2 PI: '+inst+' Node 1'
            lst.append(xyPlot.XYDataFromHistory(odb=odb, name='R2colBot-'+
                a+n,
                outputVariableName=name))


    #=========== Individual columns ============#
    for col in lst:
        func.XYplot(modelName,
        plotName =col.name[1:], # "1:"" to not include "_" added by abaqus
        xHead='Time [s]', yHead='Force [N]',
        xyDat=col)


    #=========== Total force ============#
    tpl=tuple(lst)
    #Compine all to one xyData
    xyR2 = sum(tpl)
    #Plot
    func.XYplot(modelName,
        plotName='R2colBot',
        xHead='Time [s]', yHead='Force [N]',
```

```
        xyDat=xyR2)




def xyForceDisp(modelName, x, z):
    odb=func.open_odb(modelName)

    #=========== R2 at column base  ============#
    #Create xy data for each col base
    alph = map(chr, range(65, 65+x)) #Start at 97 for lower case letters
    numb = map(str,range(1,z+1))
    count = 0
    lst=[]
    for a in alph:
        for n in numb:
            count = count + 1
            inst = 'COLUMN_' + a + n + "-1"
            name='Reaction force: RF2 PI: '+inst+' Node 1'
            lst.append(xyPlot.XYDataFromHistory(odb=odb,
                outputVariableName=name))
    tpl=tuple(lst)
    #Compine all to one xyData
    xyR2 = sum(tpl)
    #Plot
    func.XYplot(modelName,
        plotName='R2colBase',
        xHead='Time [s]', yHead='Force [N]',
        xyDat=xyR2)


    #=========== U2 at center slab  ============#

    xyU2 = xyPlot.XYDataFromHistory(odb=odb, outputVariableName=
        'Spatial displacement: U2 PI: SLAB_A1-1 Node 61 in NSET CENTERSLAB
            ',
        name='xyU2')
    func.XYplot(modelName,
        plotName='U2centerSlab',
        xHead='Time [s]', yHead='Displacement [mm]',
        xyDat=xyU2)


    #=========== Force-Displacement  ============#
    xyRD = combine(-xyU2,xyR2)
    func.XYplot(modelName,
        plotName='forceDisp',
        xHead='Displacement [mm]', yHead='Force [N]',
        xyDat=xyRD)




def xyUtopCol(modelName, column):
    '''
    Prints U1, U2 and U3 at top of column.
```

```
    modelName     = name of odb
    column        = name of column that is removed in APM
    printFormat = TIFF, PS, EPS, PNG, SVG
    stepName      = name of a step that exist in the model
    '''


    #Open ODB
    odb = func.open_odb(modelName)

    #Find correct node number and name of column
    nodeSet = odb.rootAssembly.instances[column].nodeSets['COL-TOP']
    nodeNr = nodeSet.nodes[0].label

    u1Name ='Spatial displacement: U1 PI: '+column+' Node '+str(nodeNr)+\
        ' in NSET COL-TOP'
    u2Name ='Spatial displacement: U2 PI: '+column+' Node '+str(nodeNr)+\
        ' in NSET COL-TOP'
    u3Name ='Spatial displacement: U3 PI: '+column+' Node '+str(nodeNr)+\
        ' in NSET COL-TOP'

    #Create XY-Data
    xyU1colTop = xyPlot.XYDataFromHistory(odb=odb,
        outputVariableName=u1Name, suppressQuery=True, name='U1colTop')
    xyU2colTop = xyPlot.XYDataFromHistory(odb=odb,
        outputVariableName=u2Name, suppressQuery=True, name='U2colTop')
    xyU3colTop = xyPlot.XYDataFromHistory(odb=odb,
        outputVariableName=u3Name, suppressQuery=True, name='U3colTop')

    func.XYplot(modelName, plotName='U1colTop',
        xHead ='Time [s]',
        yHead = 'Displacement [mm]',
        xyDat= xyU1colTop)
    func.XYplot(modelName, plotName='U2colTop',
        xHead ='Time [s]',
        yHead = 'Displacement [mm]',
        xyDat= xyU2colTop)
    func.XYplot(modelName, plotName='U3colTop',
        xHead ='Time [s]',
        yHead = 'Displacement [mm]',
        xyDat= xyU3colTop)


def xyAPMcolPrint(modelName, column):
    '''
    Prints U2 at top of removed column in APM.

    modelName     = name of odb
    column        = name of column that is removed in APM
    printFormat = TIFF, PS, EPS, PNG, SVG
    stepName      = name of a step that exist in the model
    '''


    #Open ODB
    odb = func.open_odb(modelName)
```

```

    #Find correct node number and name of column
    nodeSet = odb.rootAssembly.instances[column].nodeSets['COL-TOP']
    nodeNr = nodeSet.nodes[0].label
    varName ='Spatial displacement: U2 PI: '+column+' Node '+str(nodeNr)+\
        ' in NSET COL-TOP'

    #Create XY-curve
    xyU2colTop = xyPlot.XYDataFromHistory(odb=odb, outputVariableName=
        varName,
        suppressQuery=True, name='U2colTop')

    func.XYplot(modelName, plotName='U2colTop',
        xHead ='Time [s]',
        yHead = 'Displacement [mm]',
        xyDat=xyU2colTop)
```