SCHOOL OF CIVIL ENGINEERING

GENERALISED COMPONENT METHOD-BASED FINITE ELEMENT ANALYSIS OF STEEL FRAMES

RESEARCH REPORT R969

S YAN
K J R RASMUSSEN

June 2021

ISSN 1833-2781

Copyright Notice

School of Civil Engineering, Research Report R969
Generalised Component Method-based finite element analysis of steel frames
S Yan PhD
K J R Rasmussen MScEng, PhD, DEng
June 2021

ISSN 1833-2781

This publication may be redistributed freely in its entirety and in its original form without the consent of the copyright owner.

Use of material contained in this publication in any other published works must be appropriately referenced, and, if necessary, permission sought from the author.

Published by:
School of Civil Engineering
The University of Sydney
Sydney NSW 2006
Australia

ABSTRACT

This report presents a new analysis approach that incorporates macro-element joint models based on the Generalised Component Method in the FE analysis of steel frame buildings. The new analysis approach is termed Generalised Component Method-based finite-element (GCM-FE) analysis. The fundamental aspects and principles of the GCM-FE analysis approach are established in this report, including the framework of GCM-FE analysis, the constitutive models for the connection components and the implementation of GCM-FE analysis in commercial numerical software including an automatic modelling technique. The GCM-FE joint modelling method is first validated against the experimental results of three steel beam-to-column connection types, including the bolted moment end-plate connection, top-and-seat angle connection and web angle connection. GCM-FE analysis is subsequently performed on a two-storey four-bay irregular steel frame, showing apparent advantages over the traditional analysis methods which adopt simplified joint models. The GCM-FE analysis not only provides the ultimate resistance and failure mode of the frame, but also accurately predicts the load-redistribution process inside the connections and the resultant effect on the structural framework.

KEYWORDS

Direct Design Method, advanced analysis, Component Method, steel frame analysis, steel connections

Table of Contents

GENERALISED COMPONENT METHOD-BASED FINITE ELEMENT ANALYSIS OF STEEL FRAMES 5

1. Introduction 5
2. GCM-FE analysis approach 6
   - 2.1. Framework of GCM-FE analysis 7
   - 2.2. Constitutive models for components 10
3. Implementation of GCM-FE analysis in Abaqus 13
   - 3.1. Modelling of joints and members 13
   - 3.2. Automatic modelling using Python scripts 14
4. Validation of GCM-FE joint model 15
   - 4.1. Bolted moment end-plate connections 15
   - 4.2. Top-and-seat angle connection 19
   - 4.3. Web angle connections 21
5. Frame analysis example 23
6. Conclusions 26
7. Acknowledgement 26
8. References 26
9. Appendix A 29
10. Appendix B 32
11. Appendix C 69

GENERALISED COMPONENT METHOD-BASED FINITE ELEMENT ANALYSIS OF STEEL FRAMES

Shen Yan[1] **and Kim J.R. Rasmussen**[2]

[1] *College of Civil Engineering, Tongji University, Shanghai, 20092, China*
[2] *School of Civil Engineering, The University of Sydney, Sydney, NSW 2006, Australia*

1. Introduction

Current standards for the design of steel structures are premised on a two-step approach in which a structural analysis is first performed to obtain internal actions (e.g. bending moments and axial forces) and the strengths of each member and connection are subsequently checked to provisions stipulated in the standard. This conventional member-based approach is a product of past limitations on available structural analysis programs and computer hardware [1]. However, over the last decade, advances in computing power and analysis software have made it possible to design steel structural frames by computer without recourse to a structural standard for individual member checks [2-4]. This design-by-analysis approach lays the foundation for the next generation of structural design standards in which the strength and structural safety check are performed in a single step at system level.

The design-by-analysis methodology is permitted in the Australian Steel Structures Standard AS4100 [5] and in the American counterpart ASIC-360 [6], and is termed the Direct Design Method in [7]. While the behaviour of members, like columns and beams, in a structural framework is well understood and well-tested, and nonlinear analysis software is becoming increasingly available for predicting member strength and collapse behaviour, current modelling of structural steel frameworks employs highly idealised models for connections (joints), which are typically assumed to be simple (pinned) or rigid, treated as deterministic, and assumed not to fail. It is implicit that the connection strength needs to be checked independently to a structural standard and needs to be conservative. It is well-known that semi-rigidity in joints can severely affect the strength and serviceability of unbraced steel frames [8], and hence needs to be incorporated in accurate advanced analysis (or, in European terminology, "GMNIA"). However, to date, no attempt has been made towards considering the failure of joints in the structural analysis, without which the complete set of limit states cannot be accurately predicted. Incorporating the real behaviour of steel connections in the structural analysis will also produce a method of design with more uniform reliability of the structural system.

Joints feature complex geometries and are difficult to model in finite element analysis. At present, it is not a practical proposition to discretise each component of joints in routine design but rather, models that capture the stiffness and strength at member action level are more common, typically moment-rotation response models. Several types of models can be used to determine the mechanical behaviour of joints, e.g. [9] provides an overview of available analytical, empirical, experimental and mechanical models. Component-based mechanical models have been developed by several researchers for the prediction of moment-rotation curves for a wide range of connections, where the number of physical governing parameters is limited [10].

The Component Method divides a connection into a system of springs, each representing a component of the connection transferring actions. Early versions of the Component Method [11, 12] were developed into the format that is now implemented in EN 1993-1-8 [13], and allows the initial semi-rigid stiffness and flexural resistance of joints to be calculated. Bilinear spring models were later presented for modelling both the initial elastic response of a spring and the subsequent inelastic response [14, 15]. Recent research at the University of Sydney has extended the use of the Component Method to the ultimate state and the post-ultimate range of steel joints, where the ultimate state can be due to either fracture of components in the tension zone or buckling in the compression zone [16]. The proposed model has been confirmed as an accurate tool for studying the full-range behaviour of steel connections. The moment versus rotation responses of steel connections can also be predicted by component based finite element method, in which the inelastic behaviour of steel plates is allowed for through material nonlinearity, and the behaviour of components, e.g. bolts, and welds, is treated by introducing nonlinear springs [17-18].

With the flexural behaviour of steel connections obtained from empirical and mechanical joint models, the analysis and design of steel frames can be performed allowing the semi-rigidity of steel connections to be considered [19-21]. However, as previously stated, most joint models are incapable of predicting the failure of joints, and hence, the connection strength needs to be checked independently. Moreover, it is observed in tests that axial forces in beams have a non-negligible influence on the flexural behaviour of steel connections, affecting both the stiffness and resistance [22-24]. In response, component-based joint models that account for the influence of axial force have been developed [10, 25]. These models generally lead to additional iteration in dimensioning connections and members because of the added interaction between connections and their adjoining members. An effort was recently made [26] to develop FEM macro-elements-based connection models and implement the models directly in the OpenSees frame analysis. However, linear constitutive models were assumed for the components of the connections, implying linear flexural connection behaviour, and therefore, the frame analysis could not predict the ultimate state when caused by connection failure.

This study presents a finite-element analysis framework that directly allows for the full-range behaviour of steel beam-to-column joints in the analysis of steel frames, by incorporating Generalised Component Method-based joint models in the numerical model. All components in each connection are explicitly modelled in the analysis and are assigned with full-range constitutive models. This new analysis methodology is termed the Generalised Component Method-based finite element (GCM-FE) analysis of steel frames. The GCM-FE analysis approach enables the complete single-step design-by-analysis approach to be performed, in which the effect of axial forces in beams and the semi-rigidity and failure of connection are explicitly considered. The direct GCM-FE approach guarantees more uniform structural system reliability, and provides a greater understanding of the behaviour of the structural system and its mode of failure.

Seeing that creating all the connection components in the numerical model is labour-intensive and error-prone for steel frames with a large number of connections, this study further presents a method for the fast and automatic creation of GCM-FE frame analysis models in the general-purpose nonlinear FE software Abaqus using Python programming. The GCM-FE analysis framework and the corresponding automatic modelling technique are validated through comparison against experiments of various types of steel beam-to-column connections. An example is also given to demonstrate the application of the GCM-FE analysis to a practical steel frame.

2. GCM-FE analysis approach

2.1. Framework of GCM-FE analysis

The concept of employing GCM-FE analysis is illustrated in Fig. 1. For the steel frame to be analysed (Fig. 1a), each beam-to-column joint is modelled using a series of rigid bars and extensional springs, distinguishing the separate sources of deformability (Fig. 1b). The vertical rigid bar in the middle is aligned with the column centreline within the connection region, and is rigidly connected to the upper and lower columns at the top and bottom ends, respectively, with all the translational and rotational degrees of freedom constrained to ensure continuity in displacements and slopes. Each side rigid bar represents the end of the respective adjacent beam, and is rigidly connected to the beam at the intersecting node. The axial springs aligned with the beam flanges and bolt rows represent the deformability of the connection components.

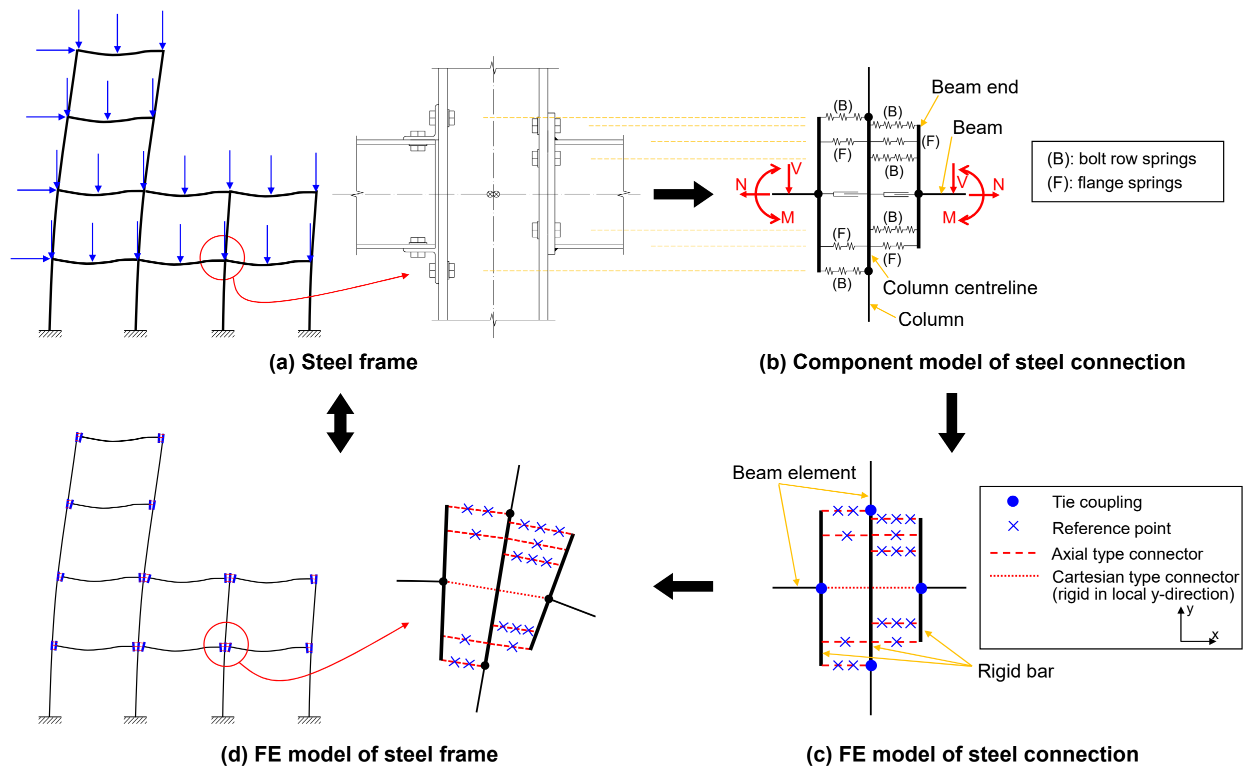


Fig. 1 Methodology of Component-Method based analysis of steel frames.

This method of decomposition and composition is generally similar to the Component Method implemented in EN 1993-1-8 [13], but differs from the Eurocode Component Method and other component-based methods [14, 27-28] in two aspects. Firstly, not only the deformability of the beam bottom flange but also that of the beam top flange is allowed for in the joint model, in order to account for the possible loading cases with reversed bending moments at the connections, which are very common for steel frames subjected to seismic loading or progressive collapse following a sudden column loss. To incorporate both beam flanges in the joint model, as well as all other sources of deformability including those of the column web panel and those of the connection, the constitutive model of each component should cover the negative-to-positive full-range behaviour, as shown in Fig. 2 and explained further in Section 2.2. For instance, the Beam flange in compression component is defined to have zero tensile stiffness, which deactivates the stiffness contribution of the beam top flange under positive bending moment (due to the separation of the beam top flange and its abutting column flange), deactivates that of the beam bottom flange under negative bending moment, and deactivates those of both flanges when the tensile force dominates the axial deformation, separating both beam

flanges from the column flange. Secondly, new connection components are proposed and incorporated in the GCM-FE analysis framework, including the Bolt slip component and the Beam flange contact component, as shown in Fig. 2. Slip of bolts is commonly observed in bolted connections, leading to non-negligible plateaus in the moment-rotation curves [23], which are not captured by the conventional Component Method models. Similarly, the conventional Component Method models are not capable of capturing the phenomenon that, in certain joint configurations such as the web angle connections tested in [29], the initially separated beam bottom flange and column flange (i.e., there is a gap between the beam end and column flange) come into contact under large joint rotation, increasing the lever arm and altering the load transferring mechanism of the connection. The two new connection components enable the GCM-FE analysis to accurately simulate the above two phenomena, and thereby capture the resultant change of load transfer path in the connections and the impact thereof on the entire frame.

To implement the GCM-FE analysis, FE software such as Abaqus and OpenSees are employed to develop the FE model, as shown in Fig. 1c. The rigid bars in the joint model can be represented by rigid elements, very stiff elastic beam-column elements which are considered as equivalent rigid elements, or rigid element type constraints. The axial springs can be represented by spring elements, connector elements, truss or link elements with appropriate axial stiffness, or link type constraints. The choice of modelling methods for the rigid bars and axial springs depends on their availability in the FE software and also on personal judgment, and therefore, varies between modellers. In this study, Abaqus instead of OpenSees is used to perform the GCM-FE analysis. This is because, as a general-purpose nonlinear FE software, Abaqus has the advantage of extending the GCM-FE approach to more complicated analyses, e.g. a hybrid simulation that uses 3D solid elements at the critical connections to more accurately capture the complex behaviour of connections including fracture. The recommended method of implementing the GCM-FE approach in Abaqus is elaborated in Section 3.

Having developed the GCM-FE model for a steel frame, the behaviour of the frame can be obtained in a computational cost-effective fashion and also with high accuracy, especially when semi-rigid connections are employed and their effect should be allowed for, as shown in Fig. 1d. The GCM-FE analysis approach has the following apparent advantages. Firstly, the full-range constitutive model is proposed (Section 2.2) for each component contributing to the deformability of the steel joint, simulating the full-range behaviour of components including gradual plasticity and component failure. The full-range component models, combined with the plasticity models defined for the structural member materials, enable the full-range behaviour of the steel frame including the ultimate resistance and failure mode to be obtained in a single run of FE analysis. Secondly, in the frame analysis the forces in all components are simultaneously determined, which through the full-range constitutive model of each component in a connection and the assembly of the component responses explicitly reflect the effect of axial force applied to the connection on its flexural behaviour, thus circumventing the need of introducing amended moment-rotation models for connections subject to the effect of axial forces. Thirdly, the full-range constitutive models proposed for the components include the behaviour of each component under both tension and compression, in contrast to the conventional component models which define the behaviour of a component under either tension or compression. This allows the GCM-FE analysis to be performed for loading conditions which generate reversed bending moments or catenary forces at the connections. Fourthly, through the two new connection components incorporated in the GCM-FE analysis framework, i.e. the Bolt slip component and the Beam flange contact component, the GCM-FE analysis approach is capable of capturing the complicated behaviour of bolted connections under large joint rotation, as well as the resultant change of load transfer path through the connection and the effect thereof on the frame response.

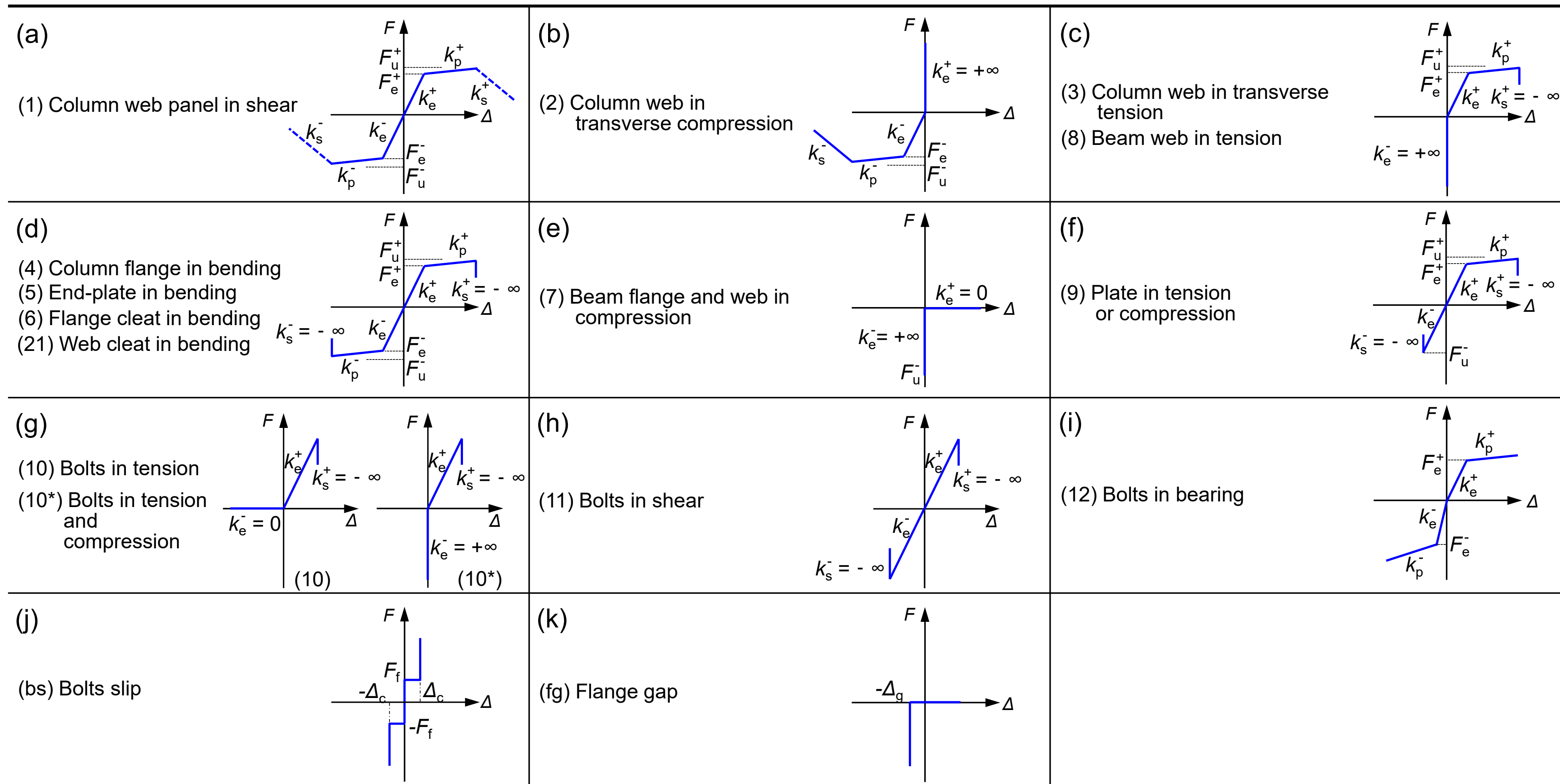


Fig. 2 Component properties

2.2. Constitutive models for components

The positive-to-negative full-range constitutive relation of each component type is a key element of the GCM-FE framework. Fig. 2 summarises the constitutive relations of all the component types in a steel beam-to-column connection, including the two new components proposed in this study, i.e. the Bolt slip component and Flange gap component (the last row in Fig. 2), as well as the components specified in EN 1993-1-8 [13]. It is noted that the Web cleat in bending component (i.e. Component 21) is a component proposed in [29], and is also included in Fig. 2. All the components are categorised into 11 groups according to their mechanical behaviour.

The following paragraphs detail the parameters defining the elastic, inelastic and post-ultimate properties of each of the component groups shown in Fig. 2. For most groups, the parameters describing the elastic range can be obtained from EN 1993-1-8, whereas other parameters defining the inelastic behaviour and ductility are obtained from the literature, where available, or simple assumptions are made, e.g. that the post-ultimate range is negligible (marked as $k_s = \pm\infty$), such as in the case of Component 10 Bolts in tension. Full-range models covering the pre- and post-ultimate inelastic stiffnesses have not yet been developed for some components. However, appropriate conservative values can be assigned to these stiffnesses based on engineering judgement until component models become available.

(a) The Column web panel in shear component (Component 1) has identical behaviour in the positive and negative regions. The (equal) stiffness (k_e^+, k_e^-) and resistance (F_e^+, F_e^-) of the initial linear range may be computed according to EN 1993-1-8 [13]. According to [30], the subsequent plastic range has a reduced stiffness (k_p^+, k_p^-) approximately equal to 4.6% of the initial linear stiffness, and in [16] was taken as 5% of the initial linear stiffness, which showed a good agreement. The component is normally assumed to have unlimited ductility [30], however, may also experience a softening range due to buckling. Accurate models for the ultimate resistance (F_u^+, F_u^-) of the component have not been established, nor have mechanical models been proposed to capture the post-buckling softening stiffness (k_s^+, k_s^-).

(b) The Column web in transverse compression (Component 2) is assumed to have infinite stiffness in the positive (tensile) region, i.e. it does not contribute to the tensile deformation. The initial compressive stiffness (k_e^-) and corresponding limit (F_e^-) may be computed according to EN 1993-1-8 [13]. Beyond F_e^-, the compressive stiffness reduces significantly due to yielding and/or buckling; however, the column web typically has inelastic post-buckling strength and the resistance can still increase before reaching the ultime capacity. In [30], the reduced inelastic stiffness (k_p^-) was suggested to be 2.3% of k_e^-, and the deformation range to the ultimate capacity was suggested as four to five times the deformation corresponding to F_e^-, which translates to an ultimate resistance (F_u^-) of $1.1F_e^-$. For unstiffened column webs which normally fail due to buckling, experimental results in [22] demonstrated that both the inelastic stiffness coefficient of 2.5% and ultimate resistance multiplier of 1.1 are conservative approximations. Ref. [31] developed a mechanical model for predicting k_p^-, F_u^- and the softening stiffness (k_s^-), according to which k_p^- and k_s^- can be approximated as 5% and -1% of the initial stiffness k_e^-, and F_u^- can be approximated as $1.4F_e^-$.

(c) The Column web in transverse tension (Component 3) and Beam web in tension (Component 8) components do not contribute to compressive deformability and, therefore, have infinite stiffness in the negative (compressive) region. The initial tensile stiffness (k_e^+) and corresponding resistance (F_e^+) of both components may be computed according to EN 1993-1-8 [13]. Beyond F_e^+, the stiffness reduces

to k_p^+ due to the yielding of material, and the component reaches the ultimate resistance (F_u^+) when the material of the component reaches the engineering ultimate strength f_u. Therefore, reasonable approximations of k_p^+ and F_u^+ are 2% of k_e^+ and $1.2F_e^+$, respectively, which correpond to a strain harding modulus of 2% of the initial elastic modulus and a tensile strength to yield stress ratio of 1.2.

(d) The Column flange in bending (Component 4), End-plate in bending (Component 5), Flange cleat in bending (Component 6), and Web cleat in bending (Component 21) components can all be analysed through the equivalent T-stub in tension component. The Component Method framework in EN 1993-1-8 [13] does not include the Web cleat in bending component. The study in [29] showed that the web cleat can be discretised into multiple segments, each corresponding to a bolt row, and each segment may be modelled as a Flange cleat in bending component; a new component index of 21 was temporarily given to the Web cleat in bending component. The equivalent T-stub is of primary importance in the framework of the Component Method and has attracted wide attention. Many models with various levels of complexity have been developed to predict its stiffness and strength, including the model in EN 1993-1-8 [13] which only predicts the initial stiffness and resistance, and more advanced models that are capable of approximately evaluating the whole force-displacement response [32-35]. With respect to the Flange cleat in bending and Web cleat in bending components, Ref. [23] has pointed out that the equivalent T-stub model does not capture the real deformation pattern, thereby leading to very conservative predictions. Accordingly, a mechanical model was proposed in [23] for the full-range behaviour of angle cleats in bending. It is noted that the components in this category fail by material fracture after sufficient plastic localisation, and therefore, the components lose load-carrying capacity as soon as the ultimate resistance is reached.

(e) The Beam flange and web in compression component (Component 7) is a key element that ensures both beam flanges can be included in the GCM-FE model. In the constitutive relation of the component, the positive region indicates separation between the beam flange and column flange, so the positive (tensile) stiffness is zero. The negative (compressive) stiffness may be assumed to be infinite according to EN 1993-1-8 [13] prior to the component resistance, which can also be computed from EN 1993-1-8 [13].

(f) The Plate in tension or compression component (Component 9) has asymmetric behaviour due to the potential buckling in the compression region. The tensile force-deformation may simply be determined from the material stress-strain relationship and the geometry of the plate. The initial compressive stiffness (k_e^-) is identical to the initial tensile stiffness (k_e^+), while the resistance (F_u^-) may be computed according to EN 1993-1-5 [36]. The plate may have post-buckling resistance and stiffness, which however may be ignored for a thin plate with its logitudinal edges not constrained.

(g) The Bolt in tension component (Component 10) shows a brittle failure mode in the positive (tensile) region, as bolts posses little ductility after reaching the ultimate strength (F_u^+). In the negative (compressive) region, the bolt does not transfer any load, and therefore, its compressive stiffness is set as zero.

For certain connection configurations (e.g. web angle connections with a gap between the beam end and column flange), the bolt rows transfer both tensile and compressive forces, the couple of which constitutes the moment resistance of the connection. The Bolt in tension component (Component 10) is not suitable for this situation and hence, a new component — Bolt in tension and compression (Component 10*) is introduced. This new component has infinite compressive stiffness and thus allows the transfer of compressive load. In this case, the overall stiffness of the bolt row is determined by other components in this bolt row, including the Bolt in bearing and Web cleat in bending components.

(h) The Bolt in shear component (Component 11) has symmetric behaviour in the positive and negative regions, both featuring a brittle failure mode. The stiffnesses and resistances may be computed according to EN1993-1-8 [13].

(i) The Bolt in bearing component (Component 12) shows an asymmetric response in the positive and negative regions because, in different bearing directions, the bolt is bearing against plates with different end distances, thereby leading to different stiffnesses and resistances. The initial stiffness (k_e^+, k_e^-) and corresponding resistance (F_e^+, F_e^-) may be computed according to EN 1993-1-8 [13]. The several failure modes of the Bolt in bearing component include excessive bearing deformation, bolt shear out, fracture of net cross-section, etc., all of which occur only after significant plastic deformation. In addition, a properly designed Bolt in bearing component normally does not control the ultimate resistance of the bolt row. Therefore, it is assumed that this component has unlimited ductility. The post elastic-limit stiffness (k_p^+, k_p^-) was taken as 5% of the initial stiffness in the joint models in [29], which showed a good agreement with experimental results.

(j) The Bolt slip component (Component BS) is not included in the traditional Component Method framework, including that standardised in EN 1993-1-8 [13]. Slip of bolts is a very common phenomenon in bolted connections using snug tight bolts, and even in the connections using slip-resistant bolts when the connections approach the ultimate resisance. The slip of bolts is especially prominent for the bolts under shear, i.e. those connecting cleats and plates to the beam flange and web. Take a top-and-seat angle connection [23] by way of example, the moment on the connection is initially transferred through static friction forces between the beam flanges and the abutting surfaces of the angle cleats; the static friction forces result from the preloading forces in the bolts (including snug tight bolts). As the moment increases, the static friction forces are overcome, and slip occurs between the beam flanges and the angle cleats. Clearances are gradually eliminated between the bolts and bolt holes, resulting in a plateau on the moment-rotation curve. Once bearing is established between bolts and edges of bolt holes, the joint is capable of sustaining increasing bending moment, and the top flange cleat starts to deform under increasing tensile force transferred through the top beam flange. EN1993-1-8 [13] and AISC 360 [6] both allow a bolt hole clearance of up to 2 mm. For common connection geometries, this amount of bolt hole clearance can lead to a plateau of up to 0.5° on the moment-rotation curve. A similar plateau can also be observed in web angle connections [29]. Therefore, to capture the bolt slip phenomenon and reflect its influence on the joint behaviour and even on the overall frame response, the new Bolt slip component is proposed. The component has symmetric behaviour in the positive and negative regions. The initial static friction load-transferring mechanism has infinite stiffness until the static friction due to bolt preloading (F_f) is overcome. The subsequent kinetic friction is generally smaller than the static friction, and for convenience, is assumed to be equal to the static friction. When the slip distance reaches the bolt hole clearance (Δ_c), bearing is established between the bolt and edge of the bolt hole. The bolt is assumed to be precisely located in the centre of the bolt hole initially, generating a gap between the bolt and the bolt hole edge which is equal to half of the bolt hole clearance at either side of the bolt. Thus, the abutting two plies (e.g. the flange cleat and beam flange) must have a relative displacement of twice the gap (which is the bolt hole clearance) before bearing can be established between the bolt and the bolt hole edge. Hence, the allowable slip distance is taken as the bolt hole clearance. As soon as the bolt hole clearance is eliminated, the Bolts slip component again has infinite stiffness, and the subsequent source of deformability comes from the Bolts in bearing and Bolts in shear components.

(k) The Flange gap component (Component FG) is another new component introduced in the GCM-FE analysis framework. In web angle connections with an initial gap between the beam end and column

flange, the compressive flange of the beam may later come into contact with the face of the column flange under large joint rotation. The beam-to-column contact produces high compressive stiffness, causing the contact point to become the new centre of compression of the joint, and increasing the lever arm between the parts of the connection transferring compression and tension. Hence, the moment resistance and the joint stiffness increased dramatically onwards [29]. Traditionally, contact between the lower beam flange and the supporting member has to be avoided because the subsequent increase of joint stiffness and resistance violates the simple joint assumption for the web angle connections, and may affect the accuracy of the analysis of the entire steel frame [8]. However, beam flange-to-column flange contact is inevitable when large rotations develop at the joint under extreme loading scenarios, and should be allowed for. For a connection with an initial gap of Δ_g between the beam end and column flange surface, the beam flange transmits no force before this gap is eliminated. Hence, the Flange gap component has zero stiffness in the range of $(-\Delta_g, +\infty)$ while has infinite stiffness at $-\Delta_g$.

Figure 2 defines the constitutive models for all components needed to perform a GCM-FE analysis. It is noted that while the model descriptions include the most fundamental properties of the components, the models are not unique. Rather, any model deemed reasonable and appropriate can be used to define the constitutive relations. Also, the GCM-FE analysis allows the ductility of a connection to be determined provided the ductility of each component is known. This is a major advantage of the GCM-FE analysis as it can be employed to ensure the structural integrity of connections is not compromised in the range of analysis.

3. Implementation of GCM-FE analysis in Abaqus

3.1. Modelling of joints and members

In developing the GCM-FE model in Abaqus, the rigid bars are represented by very stiff elastic beam elements with an area of 1.0×10^8 mm^2 and a moment of inertia equal to 1.0×10^{12} mm^4 in order to give them high axial and flexural stiffnesses, respectively. Stiff beam elements instead of analytical or discrete rigid beam elements are used because the former are available in both Abaqus/Standard (the implicit solver in Abaqus) and Abaqus/Explicit (the explicit solver in Abaqus), while the latter are only available in Abaqus/Standard.

The behaviour of each component is represented by an Axial-type connector element. Compared to the spring element and multi-point constraint, the connector element (abbreviated as "connector" herein) embedded in Abaqus is more powerful, as it provides an easy and versatile way to model many types of physical mechanisms whose geometry is discrete (i.e., node-to-node), yet the kinematic and kinetic relationships describing the connection are complex [37]. The many types of connectors can impose kinematic constraints, and include (nonlinear) force versus displacement (or velocity) behaviour in their unconstrained relative motion components with plastic behaviour, stopping and locking mechanism, as well as failure conditions defined.

Corresponding to a bolt row or a beam flange there are normally multiple connectors (i.e. components). Hence, for such bolt row or beam flange, as shown in Fig. 1c, a series of reference points, the number of which is equal to the number of connectors minus one, are created first between the rigid bars. The connectors are then created to connect each rigid bar to its adjacent reference point, representing the component next to the rigid bar, or to connect two adjacent reference points, representing internal components. For each Axial-connector, the full-range behaviour of the component it represents is defined as a nonlinear force versus displacement curve.

The many reference points in the GCM-FE model can sometimes lead to convergence problems. In this case, it is recommended that the components in the same row be assembled into an equivalent component (connector) first, thereby circumventing the need for using reference points. The force versus deformation behaviour of the assembled equivalent connector can be calculated through Eqs. (33) – (35) in Ref. [16], which is coded in MATLAB for easy application, see Appendix A.

The Axial-connectors transfer the bending moment at the beam end, through a force couple, and the axial force in the beam to the column. In addition to the bending moment and axial force, the shear force in the beam has to be transmitted to the column. Ideally, at each bolt row, a Cartesian-type connector with rigid behaviour in the vertical direction would be defined between the rigid bars, simulating the transfer of the shear force through the bolts, as shown in Fig. 3a. However, under large rotations of the connection, the rigid bar representing the beam end rotates significantly, and all the Cartesian-type connectors cannot remain horizontal at the same time, causing numerical problems. Therefore, only one Cartesian-type connector is used. It is noted that despite the vertical rigidity imposed by the Cartesian-type connector, the location of the connector does not indicate the centre of rotation of the connection because the horizontal distance between the rigid bars is not constrained at the connector. Examples in Section 4.1 will show that the Cartesian-type connector can be located at any bolt row, or aligned with the beam centreline, without leading to notable differences. Herein, the Cartesian-type connector aligns with the beam centreline, as shown in Fig. 3b and Fig. 1c.

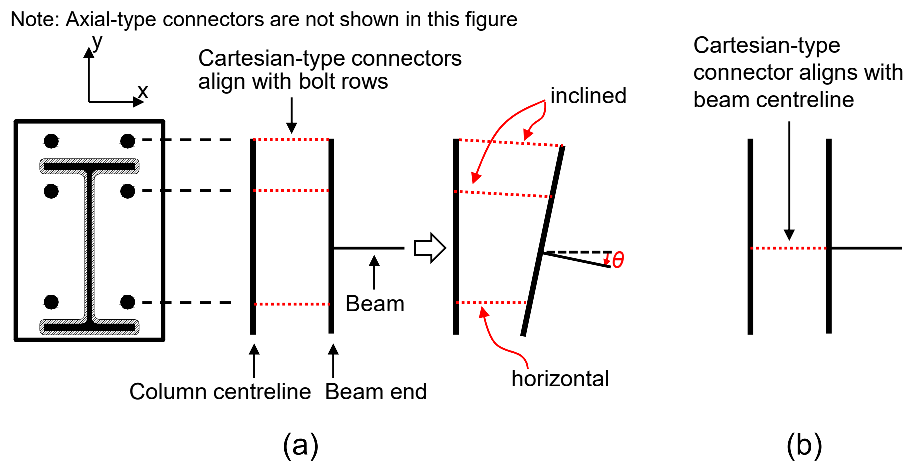


Fig. 3 Cartesian-type connector to transfer the shear force.

The beam is modelled using Beam elements (B31 element type in Abaqus), and connects to the rigid bar representing the beam end at the intersecting node through a Tie constraint, constraining all translational and rotational degrees of freedom of the rigid bar to those of the node at the beam end. Similarly, the column is also modelled using B31 elements, and connects to the rigid bar representing the column centreline at the end of the rigid bar through a Tie constraint.

3.2. Automatic modelling using Python scripts

Each connection has many components, assembled in several rows aligned with the bolt rows and beam flanges. The calculation of coordinates for all the components in every connection in the global coordinate system of the steel frame is labour-intensive and error-prone if done manually, especially for high-rise buildings that have a great number of connections with a variety of connection configurations, let alone manually creating the connectors (i.e. components) in finite-element software. This can be a prohibitive factor for the GCM-FE analysis to be widely adopted in the practical analysis and design of steel frame buildings.

To overcome this limitation and thereby promote the wide use of GCM-FE analysis, the GCM-FE modelling is automated in Abaqus through Python programming. The Python script is provided in Appendix B, enabling fast, convenient and automatic creation of GCM-FE analysis models requiring only the input of the basic information of the steel frame. The input information primarily includes four parts:

(a) frame layout, i.e. numbers of bays and stories, span of each bay, and height of each story;

(b) properties of beams and columns, including the geometric and material properties of each member profile, and the choice of profile for each beam and column;

(c) properties of connections, including the properties of each connection configuration (i.e. the distance of each row of components to the beam centreline, and the constitutive model of each component), and the connection configuration for each connection;

(d) load and boundary condition, including the types and values of the vertical and horizontal loads, and the types of boundary condition at supported nodes of the frame.

Appendix C encapsulates the input information and the format thereof to create the finite-element model with reference to a two-bay two-storey irregular steel frame (Fig. C1).

With the above input information at hand, the Python code detailed in Appendix B calculates the coordinates of all nodes for both the structural members and connectors in the connections, then creates the GCM-FE model and writes this to an input file in a format suitable for submission to Abaqus.

4. Validation of GCM-FE joint model

Having set out the framework of the GCM-FE approach and method of implementation in Abaqus, this section evaluates the proposed method of modelling joints. Joint models are developed for three joint types, including the bolted moment end-plate connection, top-and-seat angle connection, and web angle connection, and are validated against experimental results. The joint behaviour under both pure bending as well as combined bending and axial force are examined.

4.1. Bolted moment end-plate connections

The full-range behaviour of extended end-plate connections was evaluated in Ref. [22]. Two connection configurations were examined, which had identical geometric and material properties except that one (i.e. EP10 connection series) had a 10 mm thick end-plate, while the other (i.e. EP20 connection series) had a 20 mm thick end-plate and an additional backing plate. Fig. 4a shows the geometry of the connections. Both connection configurations were tested under two loading conditions, namely, bending and combined bending and axial force.

As a result of the relatively thin end plate, the EP10 connections failed in an end-plate bending failure mode, which featured significant plastic deformation of the end-plate in the tension zone and subsequent fracture initiation and propagation in the end-plate along the weld. The above failure process was reflected in the experimental moment-rotation curves as several abrupt decreases in resistance in the post-ultimate range, as shown in Fig. 5, in which the "B" curves correspond to the connection subjected to pure bending while the "B+T" curves correspond to the connection subjected to combined bending and axial tensile force. It was observed that the additional axial tensile force accelerated the failure of the end-plate in the tension zone, leading to smaller resistance and inferior joint rotation capacity. The EP20 connections had a 20 mm thick end plate which was thick enough to prevent it from failing in

bending. As a result, the connections failed due to the buckling of the unstiffened column web in the compression zone, which became the critical component of the connection. The experimental moment-rotation curves of the EP20 connections are shown in Fig. 6, in which the “B” curves correspond to the connection subjected to pure bending while the “B+C” curves correspond to the connection subjected to combined bending and axial compressive force. The additional compressive force induced earlier buckling of the column web.

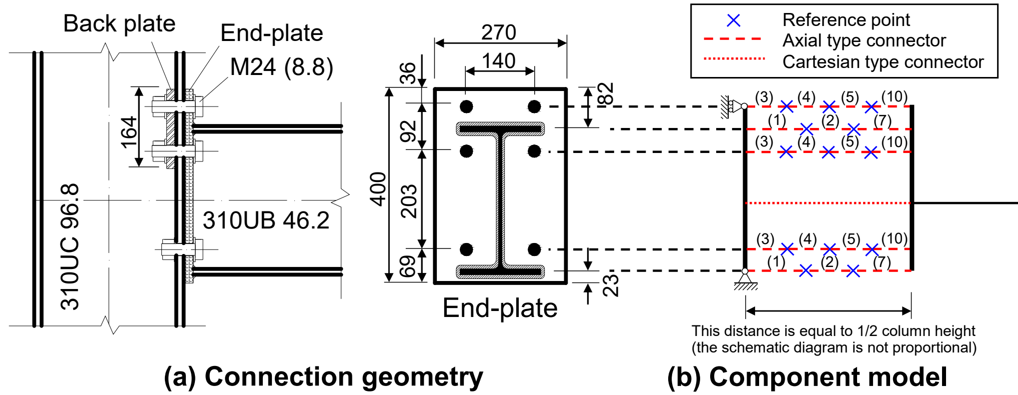


Fig. 4 Joint model for the end-plate connections in Ref. [22]

Table 1 Properties of components in EP10 connection

Spring row	h (mm)	Component	k_e^+	F_e^+	k_p^+	F_u^+	k_s^+	k_e^-	F_e^-	k_p^-	F_u^-	k_s^-
Bolt row 1	193.5	(3)	688	399	6.88	798	–∞	+∞	–	–	–	–
		(4)	521	226	26.1	624	–∞	521	–226	26.1	–624	–∞
		(5)	674	243	33.7	467	–∞	674	–243	33.7	–467	–∞
		(10)	2150	–	–	–	–∞	0	–	–	–	–∞
Bolt row 2 Bolt row 3	103.5 -103.5	(3)	688	399	6.88	798	–∞	+∞	–	–	–	–
		(4)	521	226	26.1	624	–∞	521	–226	26.1	–624	–∞
		(5)	255	265	12.8	624	–∞	255	–265	12.8	–624	–∞
		(10)	2150	–	–	–	–∞	0	–	–	–	–∞
Top flange Bottom flange	147.5 -147.5	(1)	955	705	47.8	–	–∞	955	–705	47.8	–	–∞
		(2)	+∞	–	–	–	–	1390	–563	69.5	–1039	–13.9
		(7)	0	–	–	–	–	+∞	–	–	–	–

Note: unit: h – mm; stiffness (k_e^+, k_p^+, k_s^+, k_e^-, k_p^-, k_s^-) – kN/mm; Resistance (F_e^+,F_u^+,F_e^-,F_u^-) – kN.

GCM-FE joint models are developed for the two connection configurations, as shown in Fig. 4b. The two joint models are identical in terms of active components. The components corresponding to the bolt rows are (3) column web in tension, (4) column flange in bending (the backing plate is considered as a part of the column flange), (5) end-plate in bending, and (10) bolts in tension, while the components corresponding to the beam flanges are (1) column web panel in shear, (2) column web in compression, and (7) beam flange and web in compression. The characteristics of the constitutive models for each component are as described in Section 2.2, and the model parameters are adopted as those calculated and defined in [16], in which the Generalised Component Method capable of predicting the full-range moment-rotation response of steel joints was proposed, and are validated against experimental results in [22]. Table 1 and Table 2 summarise the parameters of the constitutive model for each component considered in the GCM-FE joint models. In particular, the T-stub ultimate resistances are computed

using Swanson's T-stub model [16, 32], and the ultimate resistance of the column web in compression is determined from the experimental results in Ref. [22].

Fig. 4 and Fig. 5 show the predicted and experimental moment-rotation curves of the EP10 and EP20 connections, respectively. The GCM-FE joint models give reasonably accurate predictions of the flexural behaviour of both connection configurations, agreeing well with the experimental results in terms of the initial stiffness, ultimate resistance and rotational ductility. Especially, the joint models capture the connection failure modes, i.e. end-plate bending failure at the first bolt row in the EP10 connection and buckling of column web due to compression in the EP20 connection, and the post-ultimate responses of the connections. Moreover, the joint models are also capable of reflecting the effect of the axial force on the joint flexural behaviour.

Table 2 Properties of components in EP20 connection

Spring row	h	Comp-onent	k_e^+	F_e^+	k_p^+	F_u^+	k_s^+	k_e^-	F_e^-	k_p^-	F_u^-	k_s^-
Bolt row 1	193.5	(3)	688	399	6.88	798	–∞	+∞	–	–	–	–
		(4)	521	334	26.1	624	–∞	521	–334	26.1	–624	–∞
		(5)	5380	410	269	667	–∞	5380	–410	269	–667	–∞
		(10)	2150	–	–	–	–	0	–	–	–	–
Bolt row 2 Bolt row 3	103.5 -103.5	(3)	688	399	6.88	798	–∞	+∞	–	–	–	–
		(4)	521	334	26.1	624	–∞	521	–334	26.1	–624	–∞
		(5)	2040	452	102	624	–∞	2040	–452	102	–624	–∞
		(10)	2150	–	–	–	–	0	–	–	–	–
Top flange Bottom flange	147.5 -147.5	(1)	955	705	47.8	1410	–∞	955	–705	47.8	–1410	–∞
		(2)	+∞	–	–	–	–	1390	–563	69.5	–1039	–13.9
		(7)	0	–	–	–	–	+∞	–	–	–	–

Note: unit: h – mm; stiffness (k_e^+, k_p^+, k_s^+, k_e^-, k_p^-, k_s^-) – kN/mm; Resistance (F_e^+,F_u^+,F_e^-,F_u^-) – kN.

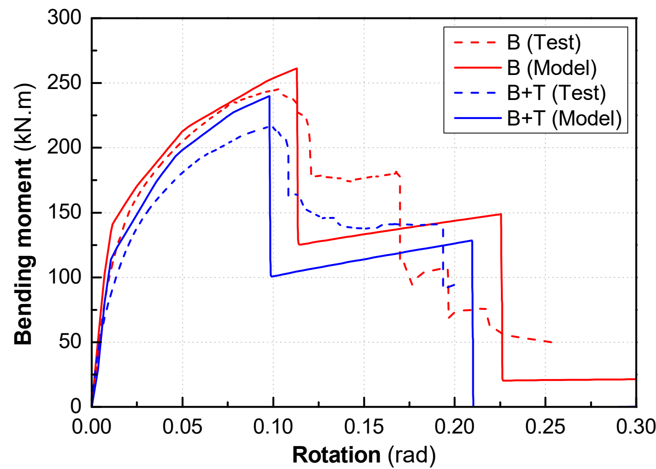


Fig. 5 Model prediction for end-plate connection EP10 in [22].

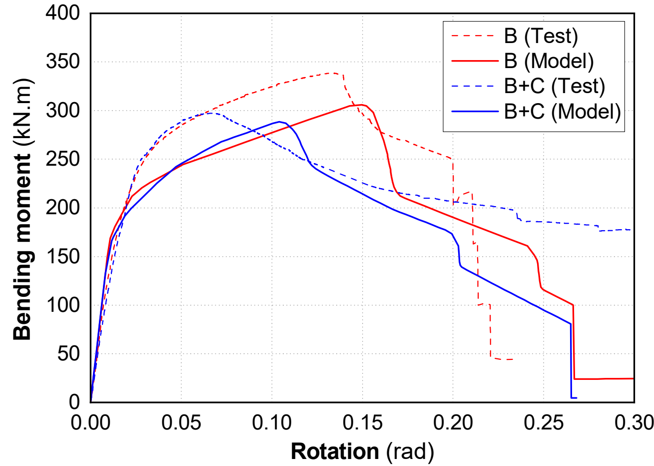


Fig. 6 Model prediction for end-plate connection EP20 in [22].

In the GCM-FE models used to produce the moment-rotation curves, the Cartesian-type connectors are aligning with the beam centreline, as explained in Section 3.1. In order to investigate the influence of different positions (heights) of the Cartesian-type connector, for instance, aligning it with one of the bolt rows, three other GCM-FE joint models are developed for the EP10 connection under pure bending moment. The moment-rotation curves predicted by the various models are shown in Fig. 7, and compared with the baseline curve obtained when the Cartesian-type connector aligns with the beam centreline, i.e. the B(Model) curve in Fig. 5. All the moment-rotation curves overlap in the initial linear range. As the joint rotation increases, the curves start to diverge slightly, with the maximum divergence observed at the ultimate resistance and in the post-ultimate range. A higher Cartesian-type connector leads to a greater ultimate resistance and a smaller joint rotational capacity but, in general, the differences between the curves are negligibly small. Comparing the three additional curves to the baseline curve, the maximum percentage differences of the resistance and joint rotation at the ultimate resistance are 2.3% and -1.2%, respectively. Therefore, as mentioned in Section 3.1, in developing the GCM-FE joint model the Cartesian-type connector is aligned with the beam centreline, which produces a near-average moment-rotation curve among all GCM-FE joint models with the connector at various heights.

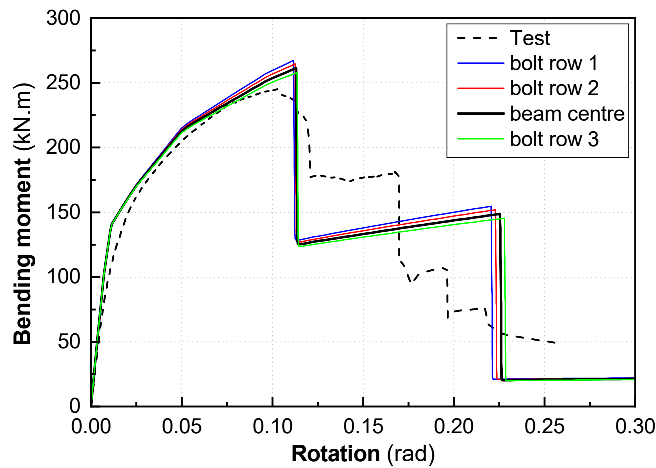


Fig. 7 Effect of assigning Cartesian-type connector to different heights.

4.2. Top-and-seat angle connection

The full-range behaviour of top-and-seat angle connections was evaluated in Ref. [23]. Two connection configurations were examined, the difference between which was the beam profile, one being 360UB 56.7 while the other being 530UB 92.4. Two loading conditions were considered in the experimental program, including pure bending and combined bending and axial tension. For both connection configurations, the tested connections had an identical top cleat bending failure mode. Hence, in this section the GCM-FE joint model is validated using only the experimental data of the connection configuration with a 360UB 56.7 beam, which was referred to as TSA360 connection in [23]. Fig. 8a shows the geometry of the connection.

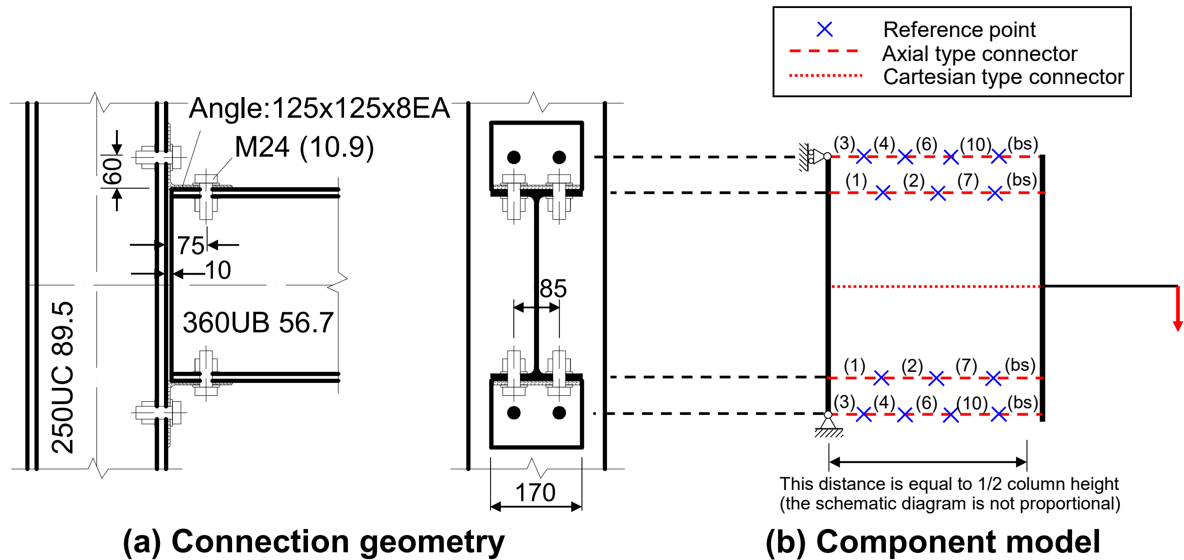


Fig. 8 Joint model for the top and seat angle connections in [23]

The GCM-FE joint model developed for the top-and-seat angle connection is shown in Fig. 8b. The components corresponding to the bolt rows are (3) column web in tension, (4) column flange in bending, (6) flange cleat in bending, (10) bolts in tension, and (BS) bolt slip, while the components corresponding to the beam flanges are (1) column web panel in shear, (2) column web in compression, (7) beam flange and web in compression, and (BS) bolt slip. The newly proposed bolt slip component is used at both the beam flanges and the bolt rows for the following reasons: At both beam flanges, the slip of bolts connecting the cleat to the flange constitutes an important source of deformation, so the bolt slip component is used at both flanges. However, under positive bending moment, the contribution of all the components aligning with the top flange, including the bolt slip component, is effectively inactivated through the use of Component 7 (i.e. the beam flange and web in compression component). Hence, an extra bolt slip component is needed at the tension zone of the connection in order to compensate for the deactivation of the bolt slip component in the top flange row, and is assigned to the top bolt row. It is noted that this bolt slip component at the top bolt row does not correspond to the behaviour of the tensile bolt that connects the top cleat to the column flange, but approximates the slip behaviour occurring at the shear bolt at the beam top flange. Another bolt slip component is used aligning with the bottom bolt row, allowing for the modelling of connection response under reversed bending moment.

The flange cleat in bending component contributes significantly to the plastic deformation in a top-and-seat angle connection, and therefore, the plasticity of this component has to be taken into account. EN 1993-1-8 provides an elastic-perfectly plastic model for angle cleats by simplifying the cleat into a T-

stub [13]. However, recent research [23] showed that the T-stub model in EN 1993-1-8 significantly underestimates the resistance of a top flange cleat by more than 85%. A mechanical model is proposed in [23] which shows significantly better accuracy than the EN 1993-1-8 T-stub model in predicting the ultimate strength, and is also capable of accurately predicting the ductility of the component. The model is adopted in the present study but modified to increase the ultimate strength by 50% as it was observed that the proposed model consistently underestimated the ultimate strength by approximately 1/3 due to several simplifying assumptions adopted, including ignoring the effect of catenary action that develops in the angle cleat under large deformation. Moreover, for the sake of simplicity, the original quadr-linear force-displacement curve is simplified into a bi-linear curve.

Table 3 Properties of components in TSA360 connection

Spring row	h	Comp-onent	k_e^+	F_p^+	k_p^+	F_s^+	k_s^+	k_e^-	F_p^-	k_p^-	F_s^-	k_s^-	Δ_c	F_c
		(3)	1250	491	–	–	–	+∞	–	–	–	–	–	–
Bolt row 1	240	(4)	8845	259	–	–	–	8845	–259	–	–	–	–	–
Bolt row 2	–240	(6)	68.1	59.0	5.67	235	–∞	68.1	–59.0	5.67	–235	–∞	–	–
		(10)	4690	527	–	–	–	0	–	–	–	–	–	–
		(BS)	–	–	–	–	–	–	–	–	–	–	2.00	55.0
		(1)	589	557	–	–	–	589	–557	–	–	–	–	–
Top flange	179.5	(2)	+∞	–	–	–	–	1360	–522	–	–	–	–	–
Bottom flange	-179.5	(7)	0	–	–	–	–	+∞	–	–	–	–	–	–
		(BS)	–	–	–	–	–	–	–	–	–	–	2.00	55.0

Note: unit: h, Δ_c – mm; stiffness (k_e^+, k_p^+, k_s^+, k_e^-, k_p^-, k_s^-) – kN/mm; Resistance (F_e^+,F_u^+,F_e^-,F_u^-,F_c) – kN.

Connections tested in this study used snug tight bolts, the tensile forces in which are not specified by design specifications and vary from application to application. Hence, the parameter F_c for the bolt slip component is not obtainable from design specifications. Here, F_c is taken as 55 kN which is an estimation of the static friction force in the tested connections. Assuming a static coefficient friction of 0.35, this value corresponds to a preloading force of 78.6 kN in each bolt, which is about 32% of the preloading force required in a slip resistant bolt. The bolt hole clearance adopted in the tested connections was 2 mm, so Δ_c in the bolt slip model is equal to 2 mm. All other components are assigned with linear elastic properties, with the initial stiffness and resistance obtained from EN 1993-1-8 [13]. Table 3 summarises the parameters of all components.

Fig. 9 shows the moment-rotation curves predicted by the GCM-FE joint model. The model performs well in predicting the initial stiffness, the gradual development of plasticity, the connection failure at the top flange cleat, and the post-ultimate response of the connection. The plateaus on the moment-rotation curves resulting from the slip of bolts are also captured. In addition, the effect of additional axial tensile force in the beam on the joint flexural behaviour, reducing both the joint resistance and rotational capacity, is also accurately predicted by the GCM-FE joint model.

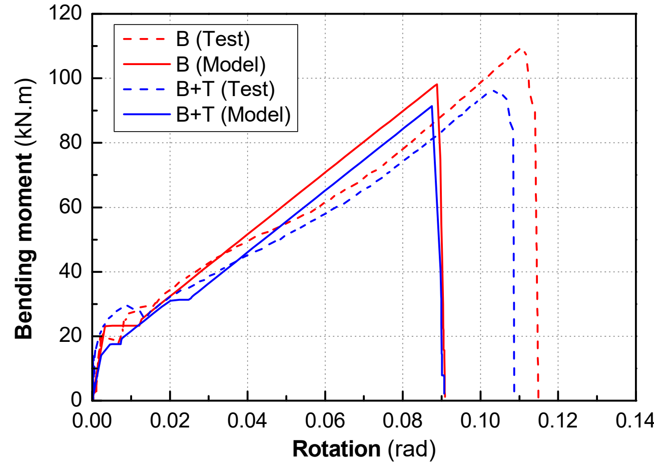


Fig. 9 Model prediction for top-and-seat angle connection TSA 360 in [23].

4.3. Web angle connections

The full-range behaviour of double web angle connections was evaluated in [29]. Two connection configurations were examined, the differences between which were the beam profile and the number of bolt rows. The first configuration had a 360UB 56.7 beam and three bolt rows, while the second configuration used a larger beam profile – 460UB 82.1 and featured two more bolt rows. The connection configurations had identical failure modes, characterised by the progressive fracture of the web cleat along the cleat root. Hence, in this section the GCM-FE joint model is validated only using the experimental data of the connection configuration with 360UB 56.7 beam, which was referred to as WA360 connection in [29]. Fig. 10a shows the geometry of the connection.

The GCM-FE joint model for the WA360 connection is shown in Fig. 10b. Compared to the models developed for the end-plate connection and top-and-seat angle connection, the web angle connection model has many more components due to the complicated transition of load-transferring mechanisms in the connection, including primarily the change of bearing state of each bolt, and the potential presence of contact between the beam bottom flange and the column flange. The components corresponding to the bolt rows are (1) column web panel in shear, (2) column web in transverse compression, (3) column web in tension, (4) column flange in bending, (10*) bolt in tension and compression, (11) bolt in shear, (12a) bolt in bearing against angle cleat, (12w) bolt in bearing against beam web, (21) web angle in bending, and (BS) bolt slip. The components corresponding to the beam flanges are (1) column web panel in shear, (2) column web in compression, (7) beam flange and web in compression, and (FG) flange gap. The model parameters are taken as those adopted in [29], and are summarised in Table 4.

Fig. 11 shows the moment-rotation curves predicted by the GCM-FE joint model. The model gives reasonably accurate predictions of the initial stiffness, the gradual development of plasticity, the connection failure and the post-ultimate response. Especially, the GCM-FE model developed for the web angle connection is capable of capturing the complicated changes of load transferring mechanisms, including the bearing state of each shear bolt, transmitting either tensile or compressive force, and the contact between the beam bottom flange and the column flange, thereby producing a full-range moment-rotation curve that reflects the above phenomena and is close to the experimental curve.

The WA360 connection was also tested under combined bending and axial tension, with the results reported in [24]. The additional axial tensile force in the beam accelerated the failure of the connection

in the tension zone, leading to a smaller ultimate resistance and reduced rotational ductility, as shown in Fig. 11. The developed GCM-FE joint model well captures these effects of additional axial load.

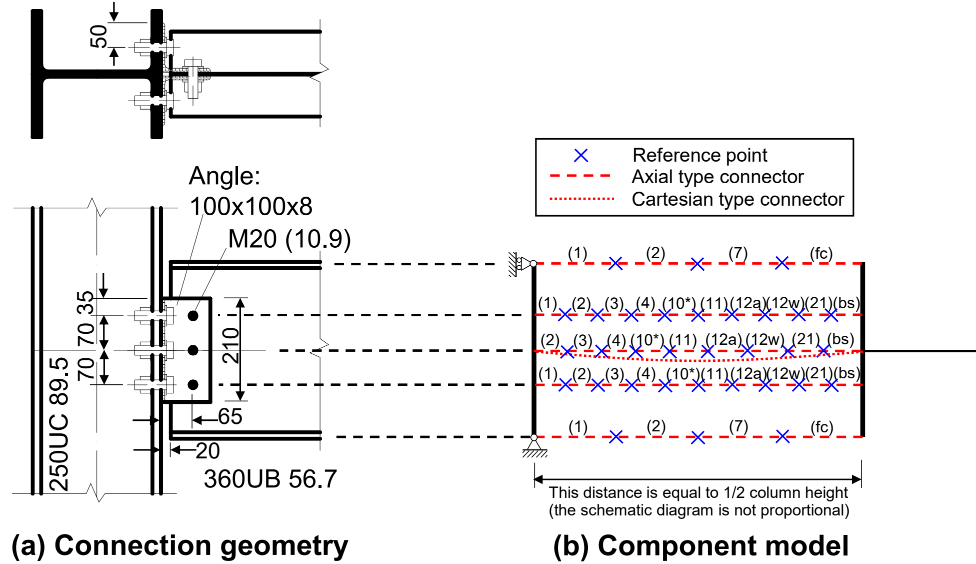


Fig. 10 Joint model for the web angle connections in [29].

Table 4 Properties of components in WA360 connection

Spring row	h	Comp-onent	k_e^+	F_e^+	k_p^+	F_u^+	k_s^+	k_e^-	F_e^-	k_p^-	F_u^-	k_s^-	Δ_c	F_c	Δ_g
		(1)#	1820	557	–	–	–	1820	–557	–	–	–	–	–	–
		(2)	+∞	–	–	–	–	1380	–519	–	–	–	–	–	–
		(3)	758	320	–	–	–	+∞	–	–	–	–	–	–	–
Bolt row 1	70	(4)	2510	298	–	–	–	2510	–298	–	–	–	–	–	–
Bolt row 2	0	(10*)	3643	441	–	–	–	+∞	–	–	–	–	–	–	–
Bolt row 3	–70	(11)	800	294	–	–	–	800	–294	–	–	–	–	–	–
		(12a)	318	209	–	–	–	424	–377	–	–	–	–	–	–
		(12w)	182	86.5	9.1	–	–	214	–127	10.7	–	–	–	–	–
		(21)	137	38.3	7.03	207	–∞	137	-38.3	7.03	-207	–∞	–	–	–
		(BS)	–	–	–	–	–	–	–	–	–	–	2.00	55.0	–
		(1)	1020	557	–	–	–	1020	–557	–	–	–	–	–	–
Top flange	179.5	(2)	+∞	–	–	–	–	1380	–519	–	–	–	–	–	–
Bottom flange	-179.5	(7)	0	–	–	–	–	+∞	–	–	–	–	–	–	–
		(fc)	–	–	–	–	–	–	–	–	–	–	–	–	20

Note: unit: h, Δ_c,Δ_g – mm; stiffness (k_e^+, k_p^+, k_s^+, k_e^-, k_p^-, k_s^-) – kN/mm; Resistance (F_e^+,F_u^+,F_e^-,F_u^-,F_c) – kN.

Bolt row does not have component (1)

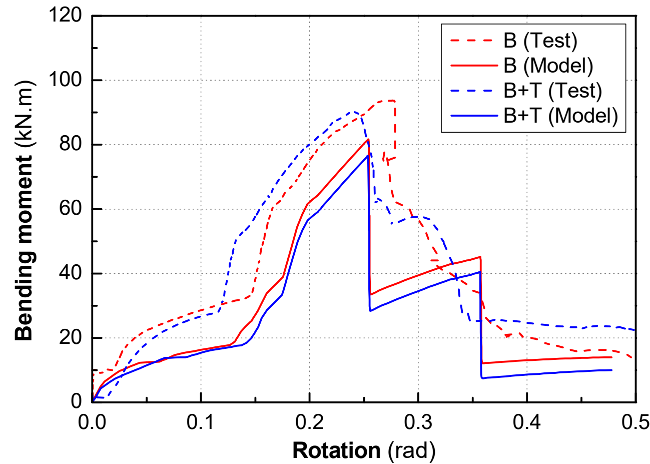


Fig. 11 Model prediction for web angle connection WA360 in [24, 29].

5. Frame analysis example

Having validated the GCM-FE joint model, a GCM-FE analysis is performed on a two-storey four-bay irregular frame, which is adopted from [19]. Fig. 12 shows the frame layout, loading conditions and member profiles. European sections (i.e. IPE and HEB sections) are used for the beams and columns, and the material is steel S275, with a modulus of elasticity of 210,000 MPa and a yield stress of 275 MPa.

Bolted moment end-plate connections are used at all beam-to-column connections. There are six connection configurations, the geometric parameters of which are given in Table 5. The flexural behaviour of each connection configuration can be evaluated from the GCM-FE joint model presented in S`ection 4.1. In developing the joint models, the model parameters for the connection components are defined as shown in Table 6. The initial stiffness and elastic limit are determined from EN 1993-1-8 [13]. The inelastic stiffnesses of the components in the compression zone (i.e. Components 1 and 2), the tensile component in the tension zone (i.e. Component 3), and the bending components in the tension zone (Components 5 and 6) are assumed as 5%, 2% and 5% of the initial stiffness of the respective components. The ultimate resistances of the components in the compression zone, the tensile component in the tension zone, and the bending components in the tension zone are assumed as 1.4, 1.2 and 1.5 times the elastic limit of the respective components. The softening stiffnesses of the components in the compression zone are assumed as –1% of the initial stiffness of the components. The above assumptions are adopted for simplicity, and the rationale behind the assumptions is explained in Section 2.2. However, evidently, any other constitutive model deemed appropriate can be used for the components in the GCM-FE analysis.

From the GCM-FE joint models developed, the full-range flexural behaviour can be obtained for each connection configuration, shown as the moment-rotation curves in Fig. 13. The initial stiffnesses, ultimate resistances, and the failure modes are summarised in Table 5. Failure of the end-plate in bending is observed in the connections with thinner end-plate, while failure of the column web, primarily at the column web in tension component in the tension zone, is observed in the connections with thicker end-plate. Compared to the end-plate bending and column web shear failure modes, the failure mode of column web in tension has inferior ductility.

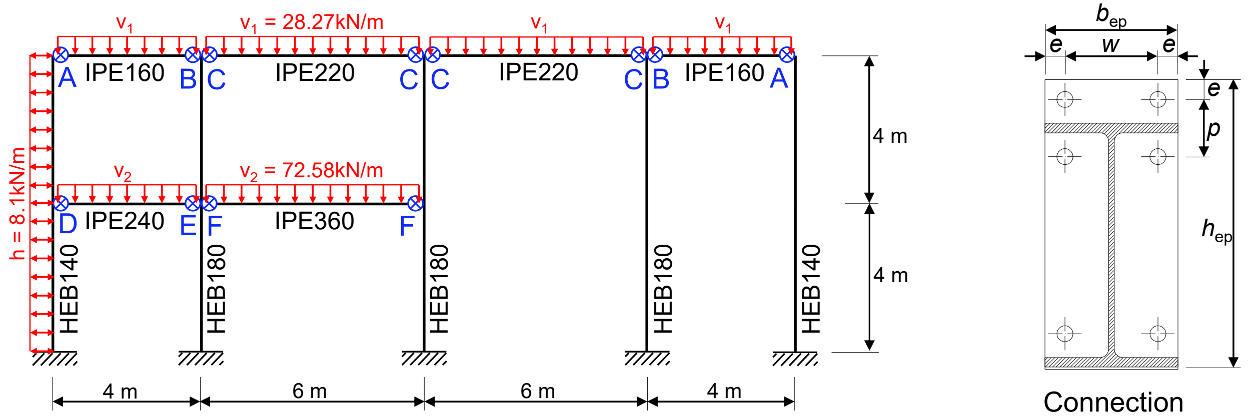


Fig. 12 Example: two-storey, four-bay irregular frame (redrawn from [19]).

Table 5 Properties of connections

Joint	Beam	Column	Bolt	h_{ep} (mm)	b_{ep} (mm)	t_{ep} (mm)	p (mm)	w (mm)	e (mm)	Stiffness (kN.m/rad)	Resistance (kN.m)	Failure mode
A	IPE160	HEB140	T16	245	140	10	90	80	30	5,900	31.8	epb
B	IPE160	HEB180	T20	240	140	12	80	80	30	8,820	39.2	cws + epb
C	IPE220	HEB180	T20	295	140	14	70	80	30	15,300	68.4	cwt
D	IPE220	HEB140	T22	295	140	16	70	80	30	11,900	51.8	cws + cwt
E	IPE220	HEB180	T22	295	140	18	70	80	30	15,800	68.6	cwt
F	IPE360	HEB180	T22	435	140	16	70	80	30	39,400	114	cwt

Table 6 Coefficients used for the constitutive models of components.

Component	k_e^+	F_e^+	k_p^+	F_u^+	k_s^+	k_e^-	F_e^-	k_p^-	F_u^-	k_s^-
(1) Column web in shear	k_e^-	$-F_e^-$	k_p^-	$-F_u^-$	k_s^-	EC3	EC3	$0.05k_e^-$	$1.4F_e^-$	$-0.01k_e^-$
(2) Column web in compression	$+\infty$	–	–	–	–	EC3	EC3	$0.05k_e^-$	$1.4F_e^-$	$-0.01k_e^-$
(3) Column web in Tension	EC3	EC3	$0.02k_e^+$	$1.2k_e^+$	$-\infty$	$+\infty$	–	–	–	–
(4) Column flange in bending	EC3	EC3	$0.05k_e^+$	$1.5F_e^+$	$-\infty$	k_e^+	$-F_e^+$	k_p^+	$-F_u^+$	$-\infty$
(5) End-plate in bending	EC3	EC3	$0.05k_e^+$	$1.5F_e^+$	$-\infty$	k_e^+	$-F_e^+$	k_p^+	$-F_u^+$	$-\infty$
(7) Beam flange and web in compression	0	–	–	–	–	$+\infty$	–	–	–	–
(10) Bolt in tension	EC3	–	–	–	$-\infty$	0	–	–	–	$-\infty$

The vertical and horizontal loads are applied proportionally on the steel frame up to and beyond the design loads shown in Fig. 12 until the complete failure of the frame. As shown in Fig. 14, the frame failed at the first beam level of the left-most bay (referred to as Beam-1-1) at a load ratio of 1.031, where the load ratio is defined as the ratio of the applied load to design load. The failure initiated at the right-hand-side connection of Beam-1-1, referred to as Conn-R. The failure of Conn-R is characterised by the progressive failure of the tensile bolt rows, from top to bottom, as shown in Fig. 14b which shows the force transferred through the bolt rows and bottom flange of Conn-R. Fig. 14c shows the bending moments in Beam-1-1 and its right and left end connections, Conn-R and Conn-L. It is observed that the failure of Conn-R results in significant reduction in the flexural resistance of the connection, which in turn leads to increases in the bending moments in both Beam-1-1 and Conn-L, triggering the

development of a plastic hinge near the centre of Beam-1-1 and the failure of Conn-L. The failure of Conn-L marks the complete failure of the steel frame.

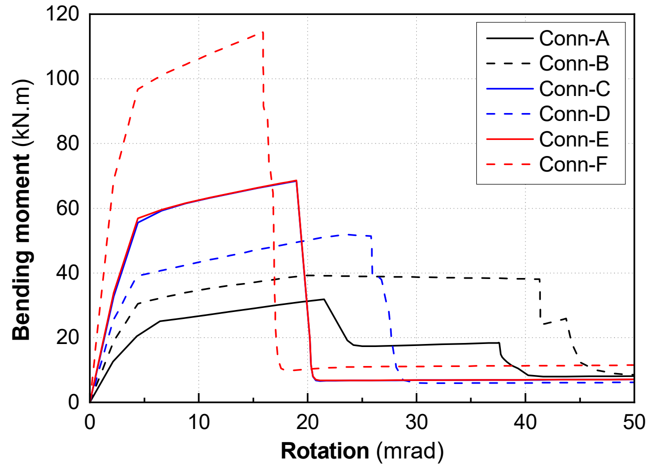
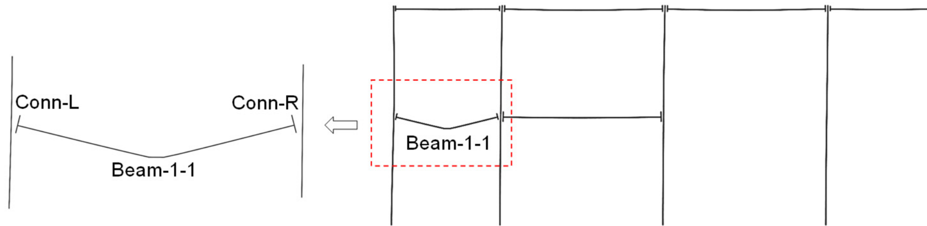
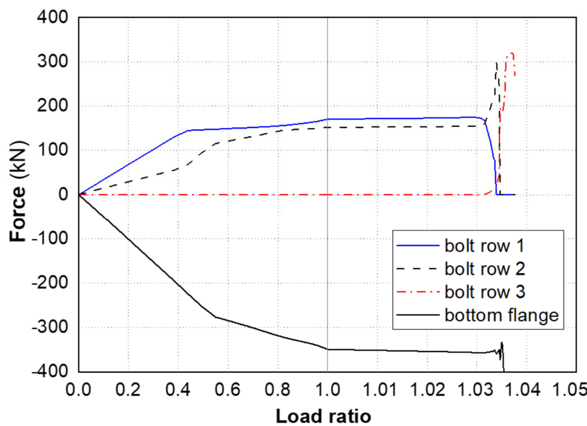


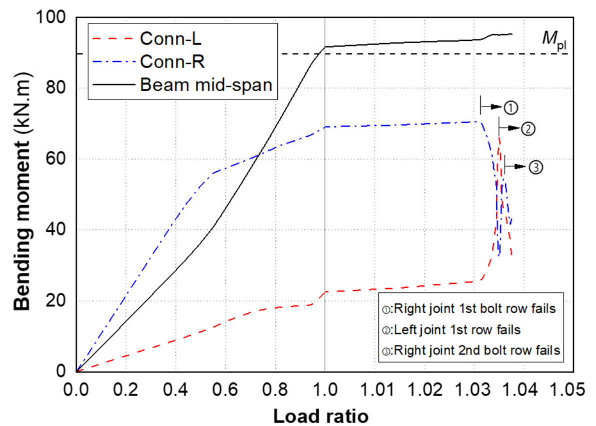
Fig. 13 Moment-rotation curves of the connections.



(a) Failure mode of the frame



(b) Force transmission in Conn-R



(c) Moment in Beam-1-1 and its end connections

Fig. 14 Full-range behaviour of the steel frame.

The observed load redistribution process demonstrates that the incorporation of the component-based connection model in the GCM-FE analysis allows the response of components inside the connections and the resultant effect on the entire structure to be captured. Moreover, Fig. 14c shows that Conn-R failed at the bending moment of 70.5 kN·m, which is higher than the ultimate flexural resistance (i.e. 68.4 kN·m) shown in Table 5 and Fig. 13. This is because the applied horizontal load induces a compressive force of 25.7 kN in Beam-1-1, postponing the failure in the tension zone of Conn-R and thereby increasing the ultimate flexural resistance of the connection. The GCM-FE analysis accurately

and explicitly allows for the effect of axial force on the flexural response of the connection, which in turn affects the response of the entire frame. This capability, and the understanding of the connection behaviour it provides, present a major advantage of the GCM-FE analysis over conventional modelling of connections at stress resultant level, typically using moment-rotation relations.

6. Conclusions

This report presents a new analysis approach that incorporates the FE macro-elements-based Component Method joint model in the FE modelling of steel frame buildings. The new analysis approach is termed Generalised Component Method-based finite element (GCM-FE) analysis. The most fundamental aspects and principles of the GCM-FE analysis approach are established in the report, including the framework of GCM-FE analysis, the constitutive models for the connection components and the implementation of GCM-FE analysis in commercial numerical software.

The GCM-FE joint modelling is the core of the GCM-FE analysis approach, and in this report, is used to produce the full-range flexural behaviour of three typical types of steel beam-to-column connections, including the bolted moment end-plate connection, top-and-seat angle connection, and web angle connection. The moment-rotation curves predicted by the GCM-FE joint models are in good agreements with the experimental results in the literature. Moreover, a GCM-FE analysis is performed on a two-storey four-bay irregular steel frame. The analysis not only obtains the ultimate resistance and failure mode of the frame, but also captures the load-redistribution process inside the connections and the resultant effect on the entire structure.

The GCM-FE analysis framework and the corresponding implementation method proposed in this study enable the complete single-step design-by-analysis approach to be performed, which is inherently more accurate and faster than the current two-step member-based design approach. It guarantees more uniform structural system reliability, and provides a greater understanding of the behaviour of the structural system and its mode of failure. Accounting for failure of both members and connections, the GCM-FE analysis approach will pave the way for introducing computer-based direct design of steel structures in the structural engineering community. Moreover, intended to explain and verify the GCM-FE analysis framework, the example and connection models in this report are all 2D. It should be noticed that the GCM-FE framework and models are readily extended to 3D.

7. Acknowledgement

The work presented in this report was funded by the Australian Research Council, via Discovery Project DP150104873.

8. References

[1] K.J.R. Rasmussen, H. Zhang. Future challenges and developments in the design of steel structures – an Australian perspective. EUROSTEEL 2017, Copenhagen, Denmark, 2017. pp. 81-94.

[2] D.W. White, A.E. Surovek, B.N. Alemdar, C. Chang, Y.D. Kim, G.H. Kuchenbecker. Stability analysis and design of steel building frames using the 2005 AISC specification. INT J STEEL STRUCT (2006), 71-97.

[3] W. Liu, H. Zhang, K.J.R. Rasmussen, S. Yan. System-based limit state design criterion for 3D steel frames under wind loads. J CONSTR STEEL RES. 157 (2019), 440-449.

[4] C. Wang, H. Zhang, K.J.R. Rasmussen, J. Reynolds, S. Yan. System reliability-based limit state design of support scaffolding systems. ENG STRUCT. 216 (2020), 110677.

[5] AS 4100, Australian Standard AS 4100 Steel Structures. Standards Australia, Sydney, Australia, 1998.

[6] AISC360-16, Specification for Structural Steel Buildings. American Institute of Steel Construction (AISC), Chicago, Illinois, 2016.

[7] H. Zhang, S. Shayan, K.J.R. Rasmussen, B.R. Ellingwood. System-based design of planar steel frames, I: Reliability framework. J CONSTR STEEL RES. 123 (2016), 135-143.

[8] J.-P. Jaspart, K. Weynand. Design of Joints in Steel and Composite Structures, ECCS – European Convention for Constructional Steelwork, 2016.

[9] C. Díaz, P. Martí, M. Victoria, O.M. Querin. Review on the modelling of joint behaviour in steel frames. J CONSTR STEEL RES. 67 (2011), 741-758.

[10] A.A. Del Savio, D.A. Nethercot, P.C.G.S. Vellasco, S.A.L. Andrade, L.F. Martha. Generalised component-based model for beam-to-column connections including axial versus moment interaction. J CONSTR STEEL RES. 65 (2009), 1876-1895.

[11] M.W. Wales, E.C. Rossow. Coupled Moment-Axial Force Behavior in Bolted Joints. J STRUCT ENG. 109 (1983), 1250-1266.

[12] F. Tschemmernegg, C. Humer. The design of structural steel frames under consideration of the nonlinear behaviour of joints. J CONSTR STEEL RES. 11 (1988), 73-103.

[13] EN 1993-1-8, Eurocode 3: Design of Steel Structures, Part 1-8: Design of Joints, Eur. Comm. Stand. Brussels, 2010.

[14] L. Simões Da Silva, A.G. Coelho, E. Lucena Neto. Equivalent post-buckling models for the flexural behaviour of steel connections. COMPUT STRUCT. 77 (2000), 615-624.

[15] L. Simões Da Silva, A. Girão Coelho. A ductility model for steel connections. J CONSTR STEEL RES. 57 (2001), 45-70.

[16] C. Zhu, K.J.R. Rasmussen, S. Yan. Generalised component model for structural steel joints. J CONSTR STEEL RES. 153 (2019), 330-342.

[17] F. Wald, L. Gödrich, L. Šabatka, J. Kabeláč, J. Navrátil. Component based finite element model of structural connections. Proceedings Steel, Space and Composite Structures, Singapore 2014, 337-344.

[18] L. Gödrich, F. Wald, J. Kabeláč, M. Kuříková. Design finite element model of a bolted T-stub connection component. J CONSTR STEEL RES. 157 (2019), 198-206.

[19] J.M. Cabrero, E. Bayo. Development of practical design methods for steel structures with semi-rigid connections. ENG STRUCT. 27 (2005), 1125-1137.

[20] M. Sekulovic, M. Nefovska-Danilovic. Contribution to transient analysis of inelastic steel frames with semi-rigid connections. ENG STRUCT. 30 (2008), 976-989.

[21] A.N.T. Ihaddoudène, M. Saidani, M. Chemrouk. Mechanical model for the analysis of steel frames with semi rigid joints. J CONSTR STEEL RES. 65 (2009), 631-640.

[22] C. Zhu, K.J.R. Rasmussen, S. Yan, H. Zhang. Experimental full-range behavior assessment of bolted moment end-plate connections. J STRUCT ENG. 145 (2019), 4019079.

[23] S. Yan, L. Jiang, K.J.R. Rasmussen, H. Zhang. Full-range behavior of top-and-seat angle connections. J STRUCT ENG. 147 (2021), 4020308.

[24] S. Yan, K.J.R. Rasmussen, L. Jiang, C. Zhu, H. Zhang. Experimental evaluation of the full-range behaviour of steel beam-to-column connections. ADV STEEL CONSTR. 1 (2020), 77-84.

[25] F. Cerfontaine, J.P. Jaspart. Analytical study of the interaction between bending and axial force in bolted joints. Third European Conference on Steel Structures (EUROSTEEL), Coimbra, Portugal, 2002.

[26] R. Costaa, F. Gentilib, L.S.D. Silvac. Simplified model for connections of steel structures in OpenSees. Nordic Steel Construction Conference 2015, Tampere, Finland, 2015. pp. 1-10.

[27] E. Bayo, J.M. Cabrero, B. Gil. An effective component-based method to model semi-rigid connections for the global analysis of steel and composite structures. ENG STRUCT. 28 (2006), 97-108.

[28] M.E. Lemonis, C.J. Gantes. Mechanical modeling of the nonlinear response of beam-to-column joints. J CONSTR STEEL RES. 65 (2009), 879-890.

[29] S. Yan, L. Jiang, K.J.R. Rasmussen. Full-range behaviour of double web angle connections. J CONSTR STEEL RES. 166 (2020), 105907.

[30] L. Simões Da Silva, A. Santiago, P. Vila Real. Post-limit stiffness and ductility of end-plate beam-to-column steel joints. COMPUT STRUCT. 80 (2002), 515-531.

[31] Chen Zhu, Full-Range Moment-Rotation Behaviour of Bolted Moment End-Plate Joints, (PhD thesis) The University of Sydney, Sydney, Australia, 2016.

[32] J.A. Swanson, R.T. Leon. Stiffness Modeling of Bolted T-Stub Connection Components. J STRUCT ENG. 127 (2001), 498-505.

[33] V. Piluso, C. Faella, G. Rizzano. Ultimate Behavior of Bolted T-Stubs. I: Theoretical Model. J STRUCT ENG. 127 (2001), 686-693.

[34] A.B. Francavilla, M. Latour, V. Piluso, G. Rizzano. Bolted T-stubs: A refined model for flange and bolt fracture modes. STEEL COMPOS STRUCT. 20 (2016), 267-293.

[35] E.G. Hantouche, A.R. Kukreti, G.A. Rassati, J.A. Swanson. Modified stiffness model for thick flange in built-up T-stub connections. J CONSTR STEEL RES. 81 (2013), 76-85.

[36] EN 1993-1-5, Eurocode 3: Design of Steel Structures, Part 1-5: Plated structural elements, Eur. Comm. Stand. Brussels, 2006.

[37] ABAQUS Inc., ABAQUS Release 6.16 Documentation, ABAQUS Inc., RI, 2016.

9. Appendix A

MATLAB file to assemble springs in series

```
%---------------------------Initialisation------------------------------%
%read from file
fid = fopen('EP20_Springs_in_series_1.txt','r');
loadType = "T";
%SM is a cell array store the infomation read from input file
SM = textscan(fid, '%f %f %f %f %f %f');
fclose(fid);

%Put SM into OS
NSM(1) = size(SM,2);    %number of parameters for each spring
NSM(2) = size(SM{1},1); %number of springs
NSPRINGS = NSM(2);

for iSM = 1:NSM(2)
   for jSM = 1:NSM(1)
      S(jSM,iSM)= SM{jSM}(iSM);
   end
end

%Put OS into corresponding matrix
%Dissemble OS into corresponding stiffness array
ke(NSPRINGS) = 0;
kPO(NSPRINGS) = 0;
kSO(NSPRINGS) = 0;
Pp(NSPRINGS) = 0;
Ps(NSPRINGS) = 0;
Pu(NSPRINGS) = 0;

for i = 1:NSPRINGS
   ke(i) = S(1,i);
   kPO(i) = S(2,i);
   kSO(i) = S(3,i);
   Pp(i) = S(4,i);
   Ps(i) = S(5,i);
   Pu(i) = S(6,i);
end

%Get preloaded spring stiffness
kp(NSPRINGS) = 0;    %preloaded plastic spring
ks(NSPRINGS) = 0;    %preloaded softening spring
for i = 1:NSPRINGS
   kp(i) = kPO(i)*ke(i)/(ke(i)-kPO(i));
   ks(i) = kSO(i)*ke(i)/(ke(i)-kSO(i));
end

%----------------Equivalent elastic spring stiffness, keqe----------------%
sumkeqe = 0;
for i = 1:NSPRINGS
   sumkeqe = sumkeqe + 1/ke(i);
end
keqe = 1/sumkeqe;
```

```
%---------------Activation sequence of preloaded springs----------------%
%determine the effective softening spring which triggers softening effect
%sort the softening critical loads and locate the smallest
%Pssorted(1): smallest softening critical load
%IndexPs(1): Index of target softening spring
[Pssorted,IndexPs] = sort(Ps);

%Put all plastic spring's critical loads into a series
for i = 1:NSPRINGS
   P(i) = Pp(i);
end
%Put the critical loads of effective softening spring at the end
NPsorted = NSPRINGS + 1;
P(NPsorted) = Pssorted(1);

%sort the critical loads array and we will have an Index array for
%activation sequence
[Psorted,IndexP] = sort(P);
%locate the critcal loads of the softening spring, those plastic springs
%with larger critical loads may not be involved into the F-D behaviour.
for i = 1:NPsorted
   if (IndexP(i) == (NPsorted))
      PLAI = i; %the index of the last activated spring
   end
end

%According to the activation sequence, assemble a stiffness array to store
%the active plastic spring stiffness in each stage.

K(1) = keqe;   %put stiffness of the equivalent elastic spring into K first
if (PLAI > 1)
   for i = 1:(PLAI-1)
      K(i+1) = kp(IndexP(i));
   end
else
   error('PLAI < 1');
end

NK = PLAI+1;    %Total number of elements in K
K(NK) = ks(IndexPs(1));

%Stiffness array of the whole spring series in each stage.Output
Kseries(NK) = 0;
%Preloaded displacement array of the whole spring series in each stage.Output
Dpreloaded(NK) = 0;
%Displacement limit array of the whole spring series in each stage.Output
dLimit(NK) = 0;

%Assemmble the series stiffness array for output
Kseries(1) = K(1);

%-----------------Displacement limit array, dLimit----------------------%
%elastic stage, preloaded displacement is zero
Dpreloaded(1) = 0.0;
%the end of elastic stage is that the first preloaded spring is active
```

```
dLimit(1) = Psorted(1)/keqe;

%displacement limit at other critical load at pre-ultimate stages.
%eqn.(19-21)
if (PLAI > 1)
   for j = 2:PLAI
      %Calculate K and preloaded displacement
      sumK = 0.0;
      sumP = 0.0;
      for i = 2:j
         sumK = sumK + 1/K(i);
         sumP = sumP + Psorted(i-1)/K(i);
      end
      %Store K
      Kseries(j) = 1/(1/K(1)+sumK);
      %Calculate displacement and store them
      Dpreloaded(j) = sumP;
      %isolate displacement
      dLimit(j) = Psorted(j)/Kseries(j)-Dpreloaded(j);
   end
else
   error('PLAI <= 1'); %in case of no plastic spring. I think allowing
                       %at least one plastic springs involving in to
                       %spring series behavour is essential for ductility
end

%displacement in post ultimate stage
%clean temporary variables
sumP = 0.0;
sumPs = 0.0;
%eqn.(30-32)
if (PLAI > 1)
   for i = 2:PLAI
      sumP = sumP + Psorted(i-1)/K(i);
      sumPs = sumPs + Psorted(PLAI)/K(i);
   end
else
   error('PLAI <= 1'); %in case of no plastic spring.
end

%Store them into output arrays
Kseries(NK) = 1/(1/K(1)+1/K(NK));
Dpreloaded(NK) = sumP - sumPs + Psorted(PLAI)/K(NK);
dLimit(NK) = Pu(IndexPs(1))/Kseries(NK) - Dpreloaded(NK);

%Pcirtical stores all the critiacal loads in differemt stage. Output
Pcritical = Psorted;
Pcritical(NK) = Pu(IndexPs(1));
%test if the post-ultimate behaviour is brittle
%brittle means that the spring series can not provide further displacement
%at softening stage. Instead, the spring series bounce back due to large
%unloading displacement at this stage.
if(dLimit(NK) < dLimit(NK-1))
   NK = NK-1;  %Brittle softening will be ignored.
end
```

```
%Put all the stored information into output matrix,
%the longitudinal Spring Object LS
%LS(NK,4) = 0;

for i = 1:NK
   LS(i,1) = Kseries(i);
   LS(i,2) = dLimit(i);
   LS(i,3) = Pcritical(i);
   LS(i,4) = Dpreloaded(i);
end

dis(1) = 0;
force(1) = 0;
for i = 1:NK
   dis(i+1) = LS(i,2);
   force(i+1) = LS(i,3);
end

plot(dis,force)

A = [force;dis];
fileID = fopen('connection_properties.txt','w');
for i = 1:NK+1
   if i == 1
      formatSpec = '(%.0f,%.2f)';
   else
      formatSpec = '*(%.0f,%.2f)';
   end
   if loadType == "T"
      fprintf(fileID, formatSpec,force(i),dis(i));
   elseif loadType == "C"
      fprintf(fileID, formatSpec,-1.0*force(NK+2-i),-1.0*dis(NK+2-i));
   else
      error('loadType is wrong');
   end
end

fclose(fileID);
%--------------------------End---------------------------------%
```

10. Appendix B

Python code to create GCM-FE model in Abaqus

```
#
# FileName: Steel frame.py
# Function: Create a steel frame, either planar or spatial, with each of the beam-to-column
#                    joints modelled based on the Component Method, using rigid bars and connectors (springs)
# this version creates irregular-shaped frame, user has to input the profiles for each beam and column,
#                    and the type/properties for each spring row
```

```
#                       each spring row contains several component, so user only need to input the
properties for each component
from abaqus import *
from abaqusConstants import *
from caeModules import *
#
#-------------------------------------------------------------------------------------------------------------------
#------------------------------------------Input Model information----------------------------------------------
-
#
# 'input_frame.txt' contains properties of all beam and column sections, including geometry and material
properties
#                                       contains properties of all connection types, including moment-
rotation behaviour of each joint
#                                       contains the layout of the frame,
#                                       contains the choice of sections for each beam/column, and the
connection type for each connection
file_input = 'input_frame.txt'
#
#-------------------------
# Constant
# Material properties
poissonsRatio = 0.3
densitySteel = 7.85*10**(-9)
largeYoungsModulus = 205000*1000.0
#
#------------------------------------------End of Input Model variables------------------------------------------
----
#---------------------------------------------------------------------------------------------------------------------
#---------------------------------------------------------------------------------------------------------------------
#---------------------------------------------------------------------------------------------------------------------
#-------------------------------------------------Initiation-------------------------------------------------------
with open(file_input) as f:
	for line in f:
		if line.startswith('Model_name'):
			list_line = [elt.strip() for elt in line.split('/')]
			model_name = str(list_line[1])
		elif line.startswith('Analysis_mode'):
			list_line = [elt.strip() for elt in line.split('/')]
			analysisMode = str(list_line[1])
			if analysisMode != "static" and analysisMode != "riks":
				raise ValueError ('Error, analysis mode input is missing or wrong')
		elif line.startswith('CPU_no'):
			list_line = [elt.strip() for elt in line.split('/')]
			cpuN = int(list_line[1])

# Mdb()
myModel = mdb.Model(name = 'Model-' + model_name)
```

```
myAssembly = myModel.rootAssembly
print_Logic = 'False'
runJob = False   # if Ture, create model and run job, otherwise create the model only
rp_Logic = False
#------------------------------------------------Functions------------------------------------------------------
def unique(list1):
	unique_list = []
	for x in list1:
		if x not in unique_list:
			unique_list.append(x)
	return unique_list

#------------------------------------------------Blueprint------------------------------------------------------
# this block returns the lay-out of the frame,including:
# bayN --> int, number of bays
# bay_span --> list, the bay span of each bay
# storyN --> int, number of story
# story_height --> list, height of each story
# x_connection --> list, x-coordinate of each connection
# y_connection --> list, y-coordinate of each connection
#
bay_span = []
story_height = []
with open(file_input) as f:
	for line in f:
		if line.startswith('Bay_number'):
			list_line = [elt.strip() for elt in line.split('/')]
			bayN = int(list_line[1])
		elif line.startswith('Story_number'):
			list_line = [elt.strip() for elt in line.split('/')]
			storyN = int(list_line[1])
		elif line.startswith('Bay_span'):
			list_line = [elt.strip() for elt in line.split('/')]
			for i in range(1,len(list_line)):
				bay_span.append(float(list_line[i]))
		elif line.startswith('Story_height'):
			list_line = [elt.strip() for elt in line.split('/')]
			for i in range(1,len(list_line)):
				story_height.append(float(list_line[i]))

if len(bay_span) != bayN: raise ValueError ('Error, bay number is inconsistent for bay_span input')

if len(story_height) != storyN: raise ValueError('Error, story number is inconsistent for story_height input')

x_connection = [0.0,]
for i in range(len(bay_span)):
	x_connection.append(x_connection[i]+bay_span[i])
```

```
y_connection = [0.0,]
for i in range(len(story_height)):
	y_connection.append(y_connection[i]+story_height[i])

if print_Logic:
	print 'bayN is', bayN
	print 'bay_span is', bay_span
	print 'connection x-coordinates are', x_connection
	print 'storyN is', storyN
	print 'story_height is',story_height
	print 'connection y-coordinates are', y_connection

#-----------------------------------------------End of Blueprint-----------------------------------------------------
-
#---------------------------------------------------Profiles---------------------------------------------------------
# Profile library
# this block returns a dictionary of profile_library.
# The keys are the profiles, the values are tuples containting the geometry of each profile, in the format
of
#
				(height, width, flange thickness, web thickness)
# For instance, profile_library['460UB74'], gives, (457, 190, 14.5, 9.1)
profile_library = {}
list_profile_code = []
with open(file_input) as f:
	for line in f:
		if line.startswith('Profile'):
			line_list = [elt.strip() for elt in line.split('/')]
			profile_code = line_list[1]
			list_profile_code.append(profile_code)
			parameter_list = []
			for i in range(2,len(line_list)):
				parameter_list.append(float(line_list[i]))
			profile_library[profile_code] = tuple(parameter_list)

if print_Logic:
	print 'profile_library is', profile_library
	print 'list_profile_code is', list_profile_code

#-------------------------
# Beam and column
# this block returns two dictionaries: beamProfile_code and columnProfile_code,
#			which tell the beams and columns used in the frame, respectively.
# The keys are the stroy ('Story$d'), the values are the beams/columns used in that story
# Therefore, to know the profile of the 3nd column in Story 1, use:
#			columnProfile_code[Story1][2], which returns a profile code, such as '200UC60'
# beamProfile_code_plus and columnProfile_code_plus are based on beamProfile_code and
```

```
#                    columnProfile_code, respectively, but add one bay of fictionary beams ('NA') to
#                    each side of the frame, and one story of fictionary columns ('NA') to the top of the
frame.
#                    These two dictionaries are used later to check whether the input beam/column are
#                    consistent with the input connection
#
beamProfile_code = {}
beamProfile_code_plus = {}
columnProfile_code = {}
columnProfile_code_plus = {}
#
with open(file_input) as f:
        line_no_beam = 0
        line_no_column = 0
        for line in f:
                if line.startswith('Beam'):
                        list_line = [elt.strip() for elt in line.split('/')]
                        line_no_beam = line_no_beam + 1
                        list_profile = []
                        story_code = list_line[1]
                        if story_code != 'Story%d' %(line_no_beam):
                                raise ValueError ('Error, beam input at Story%d is
wrong' %(line_no_beam))
                        if (len(list_line)-2) != bayN :
                                raise ValueError ('Error, number of beams in Story %d is
wrong' %(line_no_beam))
                        for i in range(2,len(list_line)):
                                if (list_line[i] not in list_profile_code) and (list_line[i] != 'NA'):
                                        raise ValueError('Error, Beam %d profile at Stroy %d does not
exist in library'\

                                                                %(i, line_no_beam))
                                list_profile.append(list_line[i])
                        beamProfile_code[story_code] = tuple(list_profile)
                        beamProfile_code_plus[story_code] = list_profile
                        beamProfile_code_plus[story_code].insert(0,'NA')
                        beamProfile_code_plus[story_code].append('NA')
                elif line.startswith('Column'):
                        list_line = [elt.strip() for elt in line.split('/')]
                        line_no_column = line_no_column + 1
                        list_profile = []
                        story_code = list_line[1]
                        if story_code!='Story%d' %(line_no_column):
                                raise ValueError ('Error, column input at Story%d is
wrong' %(line_no_column))
                        if (len(list_line)-2) != bayN+1:
                                raise ValueError ('Error, number of columns in Story %d is
wrong' %(line_no_column))
```

```
                    for i in range(2,len(list_line)):
                        if (list_line[i] not in list_profile_code) and (list_line[i] != 'NA'):
                            raise ValueError ('Error, Column %d profile at Stroy %d does
not exist in library'\

                                                    %(line_no_column,i))
                        list_profile.append(list_line[i])
                    columnProfile_code[story_code] = tuple(list_profile)
                    columnProfile_code_plus[story_code] = columnProfile_code[story_code]

columnProfile_code_plus['Story%d'%(storyN+1)] = ('NA',)*(bayN+1)

if line_no_beam != storyN: raise ValueError('Error, number of story for beam profile input is wrong')

if line_no_column != storyN: raise ValueError('Error, number of story for column profile input is
wrong')

if print_Logic:
    print 'beamProfile_code is', beamProfile_code
    print 'beamProfile_code_plus is', beamProfile_code_plus
    print 'columnProfile_code is', columnProfile_code
    print 'columnProfile_code_plus is', columnProfile_code_plus

#---------------------------------------------------End of Profiles--------------------------------------------------
#-----------------------------------------------Material  properties-----------------------------------------------
-
# Elasticity and Plasticity
# this block returns a dictionary of elasticity library containing the Young's modulus of all profiles
# this block returns a dictionary of plasticity library containing the plasticity data of all profiles
# The keys are the profiles, the values are tuples of tuples containting the plasticity data
# For instance, plasticity_library['460UB74'], gives, ((350,0),(500,0.18),(800,1.0))
elasticity_library = {}
plasticity_library = {}
list_material_code = []
with open(file_input) as f:
    for line in f:
        if line.startswith('Material'):
            line_list = [elt.strip() for elt in line.split('/')]
            material_code = line_list[1]
            list_material_code.append(material_code)

            elasticity_library[material_code] = float(line_list[2])
            tuple_plasticity = tuple(s for s in line_list[3].split("*"))
            list_plasticity = []
            for j in range(len(tuple_plasticity)):
                float_pair = (tuple(float(s) for s in
tuple_plasticity[j].strip('()').replace(' ','').split(',')))
                list_plasticity.append(tuple(float_pair))
```

```
                                        plasticity_library[material_code] = tuple(list_plasticity)

if list_profile_code != list_material_code: raise ValueError('Error, profiles and plasticity input do not
match')

if print_Logic:
        print 'elasticity_library is', elasticity_library
        print 'plasticity_library is', plasticity_library

#--------------------------------------------End of Material properties-------------------------------------------
--
#-----------------------------------------------Connector properties----------------------------------------------
--
# Connection library
# this block retains a dictionary of connection_library, inside which are serveral levels of dictionaries
# 1st level: connection_code --> {'Conn-1','Conn-2', ...}, each connection code contains several spring
rows (including
#                                                                   bolt rows, and, flanges) in
the next level
# 2nd level: springRow_code --> {'boltRow-1','boltRow-2',...,'topFlange',...,}
#                                                                   each spring row code
contains the properties of this spring row, including:
# 3nd level: properties and components --> {'y-dis', 'no. of components', 'component-1','component-
2', ...,}
# Therefore, for instance, to get access to the nonlinear behaviour of the 3rd component in the 2nd bolt
row of Conn-1, use:
#                                                                   connection_library['Conn-
1']['boltRow-2']['component-3']
connection_library = {}
list_connection_code = []
with open(file_input) as f:
        for line in f:
                if (line.startswith('Conn-')):
                        line_list = [elt.strip() for elt in line.split('/')]
                        connection_code = line_list[0]
                        springRow_code = line_list[1]
                        # create dic_springRow_property, and reset it for different connections
                        if (connection_code not in list_connection_code):
                                dict_springRow_property = {}
                        list_connection_code.append(connection_code)
                        #
                        if          (springRow_code.startswith('boltRow'))          or
(springRow_code.endswith('Flange')):
                                dict_component_property = {}
                                dict_component_property['y_dis'] = float(line_list[2])
                                dict_component_property['no_component'] = int(line_list[3])
                                if dict_component_property['no_component'] != (len(line_list)-4):
```

```
                                        raise ValueError('Error, input of component number is not
consistent for '+connection_code+' '+springRow_code)
                                for i in range(4,len(line_list)):
                                        component_number = i-3
                                        tuple_component_parameter = tuple(s for s in
line_list[i].split("*"))
                                        list_component_parameter = []
                                        for j in range(len(tuple_component_parameter)):
                                                float_pair = (tuple(float(s) for s in
tuple_component_parameter[j].strip('()').replace(' ','').split(',')))
                                                list_component_parameter.append(tuple(float_pair))

        dict_component_property['component-%d' %(component_number)] =
tuple(list_component_parameter)
                        else:
                                raise ValueError('component code input is wrong')
                        #
                        dict_springRow_property[springRow_code] = dict_component_property
                        connection_library[connection_code] = dict_springRow_property

# determine the number of different of connections that has been defined in the library
connection_code_library = tuple(unique(list_connection_code))

if print_Logic:
        print 'connection_library is', connection_library
        print 'connection_code_library is', connection_code_library

#-------------------------
# Connections in this frame
# this block retains a dictionary of connection codes that are used for each connection in the frame,
# For instance, to get the code of connection used at the right end of the beam at the Story 4, Bay 1, use:
#                                       connection_code['Story4'][0][1]         note: remember
counting in Python starts from 0
connection_code = {}
with open(file_input) as f:
        line_no = 0
        for line in f:
                if line.startswith('Connection'):
                        list_line = [elt.strip() for elt in line.split('/')]
                        line_no = line_no + 1
                        list_connection = []
                        story_code = list_line[1]
                        if story_code != 'Story%d' %(line_no):
                                raise ValueError ('Error, Story%d for connection input is
wrong' %(line_no))
                        if (len(list_line)-2) != bayN + 1 :
                                raise ValueError ('Error, number of connections in Story %d is
wrong' %(line_no))
```

```
                    for i in range(2,len(list_line)):
                            str_connections_of_a_beam = \
                                    (tuple(str(s) for s in list_line[i].strip('()').replace(' ','').split(',')))
                            for j in range(len(str_connections_of_a_beam)):
                                    if          (str_connections_of_a_beam[j]          not          in
connection_code_library) \
                                            and (str_connections_of_a_beam[j] != 'NA'):
                                            raise ValueError ('Error, Connection %d at Stroy %d
does not exist in library' \
                                                                                    %(i,
line_no))
                            list_connection.append(str_connections_of_a_beam)
                    connection_code[story_code] = tuple(list_connection)

if line_no != storyN: raise ValueError ('Error, number of story for connection_input is wrong')

if print_Logic:   print 'connection_code is', connection_code

# to check whether, in the input_frame file, the 'NA' of beam/column and connection are consistent
# beamProfle_code_plus and columnProfile_code_plus are used
for i in range(1,bayN+2):
        for j in range(1,storyN+1):
                connection = connection_code['Story%d'%(j)][i-1]
                beams_at_connection            =            (beamProfile_code_plus['Story%d'%(j)][i-
1],beamProfile_code_plus['Story%d'%(j)][i])
                columns_at_connection = (columnProfile_code_plus['Story%d'%(j)][i-1],

        columnProfile_code_plus['Story%d'%(j+1)][i-1])
                if(connection==('NA','NA')       and       beams_at_connection!=('NA','NA')       and
columns_at_connection!=('NA','NA')):
                        raise ValueError ('Error, connection and beam/column NAs are inconsistent at
Story%d Bay%d' %(j,i))
                if connection != ('NA','NA'):
                        if    beams_at_connection    ==    ('NA','NA')    or    columns_at_connection
==('NA','NA'):
                                raise  ValueError  ('Error,  connection  and  beam/column  NAs  are
inconsistent at Story%d Bay%d' %(j,i))
                        if (connection[0] == 'NA' and beams_at_connection[0] != 'NA')\
                                or (connection[0] != 'NA' and beams_at_connection[0] == 'NA')\
                                or (connection[1] == 'NA' and beams_at_connection[1] != 'NA')\
                                or (connection[1] != 'NA' and beams_at_connection[1] == 'NA'):
                                raise ValueError ('Error, connection and beam NAs are inconsistent at
Story%d Bay%d' %(j,i))

#-------------------------------------------End of Connector properties-----------------------------------------
----
#-----------------------------------------------Load  information-----------------------------------------------
--
```

```
# Load in this frame
# this block retains two variables, vLoad_type and hLoad_type, each can be either 'point_load' or 'line_load',
#                                   need to specify in the input file
# this block also retains two dictionaries of the loads that are applied on the beams and columns
# For instance, to get the of line beam load applied on the beam at Story 2, Bay 1, use:
#                                   vLoad['Story2'][0]                note: remember counting in Python starts from 0
vLoad = {}
hLoad = {}
#
with open(file_input) as f:
	for line in f:
		if line.startswith('Vertical_load'):
			list_line = [elt.strip() for elt in line.split('/')]
			vLoad_type = list_line[1]
			if vLoad_type != 'point_load' and vLoad_type != 'line_load':
				raise ValueError ('Error, vLoad_type input is wrong')
			line_no_story = 0
		if line.startswith('V_load'):
			list_line = [elt.strip() for elt in line.split('/')]
			line_no_story = line_no_story + 1
			list_load = []
			story_code = list_line[1]
			if story_code != 'Story%d' %(line_no_story):
				raise ValueError ('Error, story no. for vertical load input at Story%d is wrong' %(line_no_story))
			if vLoad_type == 'point_load':
				if (len(list_line)-2) != bayN + 1:
					raise ValueError ('Error, vertical load input in Story %d is wrong' %(line_no_story))
			elif vLoad_type == 'line_load':
				if (len(list_line)-2) != bayN:
					raise ValueError ('Error, vertical load input in Story %d is wrong' %(line_no_story))
			for i in range(2,len(list_line)):
				list_load.append(list_line[i])
			vLoad[story_code] = tuple(list_load)

# Horizontal load
with open(file_input) as f:
	for line in f:
		if line.startswith('Horizontal_load'):
			list_line = [elt.strip() for elt in line.split('/')]
			hLoad_type = list_line[1]
			if hLoad_type != 'point_load' and hLoad_type != 'line_load' and hLoad_type != 'NA':
				raise ValueError ('Error, hLoad_type input is wrong')
```

```
                    line_no_story = 0

if hLoad_type != 'NA':
        with open(file_input) as f:
                for line in f:
                        if line.startswith('H_load'):
                                list_line = [elt.strip() for elt in line.split('/')]
                                line_no_story = line_no_story + 1
                                list_load = []
                                story_code = list_line[1]
                                if story_code != 'Story%d' %(line_no_story):
                                        raise ValueError ('Error, story no. for horizontal load input at
Story%d is wrong' %(line_no_story))
                                if (len(list_line)-2) != bayN + 1:
                                        raise ValueError ('Error, horizontal load input in Story %d is
wrong' %(line_no_story))
                                for i in range(2,len(list_line)):
                                        list_load.append(list_line[i])
                                hLoad[story_code] = tuple(list_load)

if print_Logic:
        print vLoad_type
        print hLoad_type
        print vLoad
        print hLoad

# to check whether, in the input_frame file, the load are applied on the beams/columns
# vertical load
if vLoad_type == 'line_load':
        for i in range(1,bayN+1):
                for j in range(1,storyN+1):
                        v_load = vLoad['Story%d'%(j)][i-1]
                        beam_for_load = beamProfile_code['Story%d'%(j)][i-1]
                        if beam_for_load == 'NA' and v_load != 'NA':
                                raise ValueError ('Error, no beam for vertical line load at Story%d,
Beam%d' %(j,i))
elif vLoad_type == 'point_load':
        for i in range(1,bayN+2):
                for j in range(1,storyN+1):
                        v_load = vLoad['Story%d'%(j)][i-1]
                        beams_at_connection = (beamProfile_code_plus['Story%d'%(j)][i-
1],beamProfile_code_plus['Story%d'%(j)][i])
                        if v_load != 'NA' and beams_at_connection[0] == 'NA' and
beams_at_connection[1] == 'NA':
                                raise ValueError ('Error, no beam for vertical point load at Story%d,
Beam%d' %(j,i))

# horizontal load
```

```
if hLoad_type != 'NA':
	for i in range(1,bayN+2):
		for j in range(1,storyN+1):
			h_load = hLoad['Story%d'%(j)][i-1]
			column_for_load = columnProfile_code['Story%d'%(j)][i-1]
			if column_for_load == 'NA' and h_load != 'NA':
				raise ValueError ('Error, no column for horizontal load at Story%d, 
Column%d' %(j,i))

#---------------------------------------------End of Load information--------------------------------------------
---
#-------------------------------------------Boundary condition information--------------------------------------
-----
bcColumn = {}
with open(file_input) as f:
	for line in f:
		if line.startswith('BC'):
			list_line = [elt.strip() for elt in line.split('/')]
			list_bc = []
			story_code = list_line[1]
			if story_code != 'Story%d' %(1):
				raise ValueError ('Error, story no. for vertical load input at Story%d is 
wrong' %(1))
			for i in range(2,len(list_line)):
				list_bc.append(list_line[i])
			bcColumn[story_code] = tuple(list_bc)

if print_Logic:
	print bcColumn

# to check whether, in the input_frame file, the bc are applied on the first story columns
for i in range(1,bayN+2):
	bc_column = bcColumn['Story%d'%(1)][i-1]
	column_code = columnProfile_code['Story%d'%(1)][i-1]
	if (column_code != 'NA' and bc_column == 'NA') or (column_code == 'NA' and bc_column != 
'NA'):
		raise ValueError ('Error, bc_column and first story column are not consistance, 
column%d' %(i))

#----------------------------------------End of Boundary condition information-------------------------------
-------
#-------------------------------------------------End of Initiation------------------------------------------------
#------------------------------------------------------------------------------------------------------------------
#------------------------------------------------------------------------------------------------------------------
#---------------------------------------------Part, material and assembly----------------------------------------
---
#------------------------------------------------------------------------------------------------------------------
```

```
#----------------------------------------------------The connections-----------------------------------------------
-
# the reference origin is taken as the beam-column intersection
# Rigid bar material and section
materialActive = myModel.Material(name='Material-rigid')
materialActive.Elastic(table = ((largeYoungsModulus,poissonsRatio),))
myModel.RectangularProfile(name='Profile-rigid bar', a=200.0, b=200.0)
myModel.BeamSection(name='Section-rigid bar',integration=DURING_ANALYSIS,
poissonRatio=0.0,
                                                profile='Profile-rigid bar', material='Material-rigid',
                                                temperatureVar=LINEAR, consistentMassMatrix=False)
# y- coordinates of the rigid bar, used for creating constraints to column
list_y_columnCentre = {}
# list_x_beamEnd: x-coordinates of the left and right beam ends, used for creating Beam
# list_y_beamEnd: y-coordinates of top and btm rigid bars
list_x_beamEnd = {}
list_y_beamEnd = {}
#
for i in range(1,bayN+2):
        no_column = i
        for j in range(1,storyN+1):
                no_story = j
                #
                # connection and column code
                conn_code = list(connection_code['Story%d' %(j)][i-1])
                conn_index = 'Story%d-Connection%d' %(j,i)
                column_code = columnProfile_code['Story%d' %(j)][i-1]
                #
                if conn_code != ['NA','NA']:
                        # connection coordinates (connection parts are created at the origin and then
translated to correct position)
                        x_offset = x_connection[i-1]
                        y_offset = y_connection[j]
                        #-------------------------
                        # connection geometries calculation
                        x_coords = []   # contains two tuples, each from one connection side (may be
two different connection types)
                        y_left_beamEnd = []
                        y_right_beamEnd = []
                        x_beamEnd = []
                        y_beamEnd = []
                        # k is the index of side. k = 0, the left side of the connection, k = 1, the right
side
                        for k in range(2):
                                # for single-sided beam-column connection, the side with no
connecting beam is assumed to
                                #        have to same type connection. This is only for create
the rigid bars the connection
```

```
if conn_code[k] == 'NA': conn_code[k] = conn_code[abs(k-1)]
#
conn_type = conn_code[k]
#
y_coords = []
# y coords of the beam end
springRowN = len(connection_library[conn_type])
y_coords.append(connection_library[conn_type]['topFlange']['y_dis'])

y_coords.append(connection_library[conn_type]['btmFlange']['y_dis'])
for h in range(springRowN-2):

y_coords.append(connection_library[conn_type]['boltRow-%d' %(h+1)]['y_dis'])

# '-2' to exclude top and btm flanges
y_coords.append(0.0)
y_coords_unique = unique(y_coords)
if k == 0: y_left_beamEnd = y_coords_unique
elif k == 1: y_right_beamEnd = y_coords_unique

# y-coordinates of the column centre line rigid bars
# list of y_coords may have duplicated elements (if the two connections are
identical), removed
#           one of the ducplicated item from the list using unique() fuction defined
previously
y_beamEnd.append(y_left_beamEnd)
y_beamEnd.append(y_right_beamEnd)
y_combined_beamEnd = y_left_beamEnd + y_right_beamEnd
y_columnCentre          =          tuple(sorted(unique(y_combined_beamEnd),
reverse=True))
# output for constraint to column
list_y_columnCentre[conn_index]=(y_columnCentre[0],y_columnCentre[-1])
# 0-top, 1-btm
#
# x- coords of the beam end
height_columnProfile = profile_library[column_code][0]
x_beamEnd.append(-0.5*height_columnProfile)
x_beamEnd.append(0.5*height_columnProfile)
# output
list_x_beamEnd[conn_index] = x_beamEnd
list_y_beamEnd[conn_index] = y_beamEnd
#
#--------------------------
# columnCentre
# sketch and part of
nameStr = 'columnCentre-%d-%d' %(i,j)
sketchActive   =   myModel.ConstrainedSketch(name='Sketch-'+   nameStr,
sheetSize=200.0)
```

```
                    pointCoords = []
                    for h in range(len(y_columnCentre)):
                        pointCoords.append((0.0,y_columnCentre[h]))
                    for h in range(len(pointCoords)-1): sketchActive.Line(point1=pointCoords[h],
point2=pointCoords[h+1])
                    partActive = myModel.Part(name='Part-' + nameStr,
dimensionality=THREE_D, type=DEFORMABLE_BODY)
                    partActive.BaseWire(sketch = sketchActive)
                    # material and section
                    edge_part = partActive.edges
                    region_part = regionToolset.Region(edges=edge_part)
                    partActive.SectionAssignment(region=region_part, sectionName='Section-
rigid bar',
                                                                        offset=0.0,
offsetType=MIDDLE_SURFACE, offsetField='',

        thicknessAssignment=FROM_SECTION)
                    partActive.assignBeamSectionOrientation(region=region_part,
method=N1_COSINES,

        n1=(0.0, 0.0, -1.0))
                    # instance
                    instance_name = 'Instance-' + nameStr
                    myAssembly.Instance(name = instance_name, part = partActive, dependent =
ON)
                    myAssembly.translate(instanceList=(instance_name, ), vector=(x_offset,
y_offset, 0.0))
                    # mesh
                    element_type=mesh.ElemType(elemCode=B31, elemLibrary=STANDARD)
                    partActive.setElementType(regions=region_part,elemTypes=(element_type, ))
                    partActive.seedPart(size=10.0, deviationFactor=0.1, minSizeFactor=0.1)
                    partActive.generateMesh()

#-----------------------------------------------End of The connections--------------------------------------------
---
#----------------------------------------------------The  beams---------------------------------------------------
-
#
# if the vertical load are applied as line loads on beams, the beam is shorter than the bay span due to the
connection width,
# so the (equivalent) applied line load should be larger than the design value.
# This amplification is realised by beam_length_factor
list_beam_length_factor = {}
for i in range(1,bayN+1):
    no_column = i
    for j in range(1,storyN+1):
        no_story = j
        #
```

```
        profile_code = beamProfile_code['Story%d'%(j)][i-1]
        left_conn_index = 'Story%d-Connection%d' %(j,i)
        right_conn_index = 'Story%d-Connection%d'%(j,i+1)
        #
        if profile_code != 'NA':
            #
            # Beam
            nameStr = 'beam-%d-%d'%(i,j)
            beam_index = 'Story%d-beam%d'%(j,i)
            left_conn_code = connection_code['Story%d'%(j)][i-1][1]
            right_conn_code = connection_code['Story%d'%(j)][i][0]
            # beam end x-coordinates
            #       if no connection at a beam end, the beam end is connected directly to
another beam instead of a beamEnd
            if left_conn_code == 'NA':
                x_left_beamEnd = x_connection[i-1]
            else:
                x_left_beamEnd          =           x_connection[i-1]           +
list_x_beamEnd[left_conn_index][1]
            if right_conn_code == 'NA':
                x_right_beamEnd = x_connection[i]
            else:
                x_right_beamEnd          =           x_connection[i]           +
list_x_beamEnd[right_conn_index][0]
            # calculate and store beam_length_factor
            bay_span = x_connection[i] - x_connection[i-1]
            beam_length = x_right_beamEnd - x_left_beamEnd
            beam_length_factor =  bay_span / beam_length
            list_beam_length_factor[beam_index] = beam_length_factor
            # beam end y-coordinates
            y_beamEnd_active =  y_connection[j]
            # sketch and part
            sketchActive   =   myModel.ConstrainedSketch(name='Sketch-'   +   nameStr,
sheetSize=2000.0)
            sketchActive.Line(point1=(x_left_beamEnd,y_beamEnd_active),

    point2=(x_right_beamEnd,y_beamEnd_active))
            partActive = myModel.Part(name='Part-' + nameStr,

    dimensionality=THREE_D, type=DEFORMABLE_BODY)
            partActive.BaseWire(sketch = sketchActive)
            # material and section
            materialActive = myModel.Material(name='Material-' + nameStr)
            youngsModulus = elasticity_library[profile_code]
            materialActive.Elastic(table = ((youngsModulus,poissonsRatio),))
            plasticityActive = plasticity_library[profile_code]
            materialActive.Plastic(table = plasticityActive)
            geometryActive = profile_library[profile_code]
```

```
                        myModel.IProfile(name='Profile-' + nameStr, l=0.5*geometryActive[0],

        h=geometryActive[0],b1=geometryActive[1],b2=geometryActive[1],
                                                t1=geometryActive[2],
t2=geometryActive[2],t3=geometryActive[3])
                        myModel.BeamSection(name='Section-
'+nameStr,integration=DURING_ANALYSIS, poissonRatio=0.0,
                                                profile='Profile-'    +    nameStr,
material='Material-' + nameStr,
                                                temperatureVar=LINEAR,
consistentMassMatrix=False)
                        edge_part = partActive.edges
                        region_part = regionToolset.Region(edges=edge_part)
                        partActive.SectionAssignment(region=region_part, sectionName='Section-' +
nameStr,
                                                        offset=0.0,
offsetType=MIDDLE_SURFACE, offsetField='',

        thicknessAssignment=FROM_SECTION)
                        partActive.assignBeamSectionOrientation(region=region_part,
method=N1_COSINES,

        n1=(0.0, 0.0, -1.0))
                        # instance
                        instance_name = 'Instance-' + nameStr
                        myAssembly.Instance(name = instance_name, part = partActive, dependent =
ON)
                        # mesh
                        element_type=mesh.ElemType(elemCode=B31, elemLibrary=STANDARD)
                        partActive.setElementType(regions=region_part,elemTypes=(element_type, ))
                        partActive.seedPart(size=200.0, deviationFactor=0.1, minSizeFactor=0.1)
                        partActive.generateMesh()
                        #
                        #-------------------------
                        # beamEndRigid
                        # sketch and part of
                        if left_conn_code != 'NA' or right_conn_code != 'NA':
                                nameStr = 'beamEnd-%d-%d'%(i,j)
                                sketchActive   =   myModel.ConstrainedSketch(name='Sketch-'   +
nameStr, sheetSize=200.0)
                                if left_conn_code != 'NA':
                                        x_left_beamEnd = list_x_beamEnd[left_conn_index]
                                        y_left_beamEnd = list_y_beamEnd[left_conn_index][1]

                                        pointCoords = []
                                        for h in range(len(y_left_beamEnd)):
                                                pointCoords.append((x_connection[i-1]          +
x_left_beamEnd[1],
```

```
y_connection[j] + y_left_beamEnd[h]))
                                        for h in range(len(pointCoords)-1):
sketchActive.Line(point1=pointCoords[h],

                                                  point2=pointCoords[h+1])
                              if right_conn_code != 'NA':
                                        x_right_beamEnd = list_x_beamEnd[right_conn_index]
                                        y_right_beamEnd = list_y_beamEnd[right_conn_index][0]

                                        pointCoords = []
                                        for h in range(len(y_right_beamEnd )):
                                                  pointCoords.append((x_connection[i] +
x_right_beamEnd[0],

          y_connection[j] + y_right_beamEnd[h]))
                                        for h in range(len(pointCoords)-1):
sketchActive.Line(point1=pointCoords[h],

                                                  point2=pointCoords[h+1])
                              partActive = myModel.Part(name='Part-' + nameStr,

          dimensionality=THREE_D, type=DEFORMABLE_BODY)
                              partActive.BaseWire(sketch = sketchActive)
                              # material and section
                              edge_part = partActive.edges
                              region_part = regionToolset.Region(edges=edge_part)
                              partActive.SectionAssignment(region=region_part,
sectionName='Section-rigid bar',
                                                                                     offset=0.0,
offsetType=MIDDLE_SURFACE, offsetField='',

          thicknessAssignment=FROM_SECTION)
                              partActive.assignBeamSectionOrientation(region=region_part,
method=N1_COSINES,

                    n1=(0.0, 0.0, -1.0))
                              # instance
                              instance_name = 'Instance-' + nameStr
                              myAssembly.Instance(name = instance_name, part = partActive,
dependent = ON)
                              # mesh
                              element_type=mesh.ElemType(elemCode=B31,
elemLibrary=STANDARD)

          partActive.setElementType(regions=region_part,elemTypes=(element_type, ))
                              partActive.seedPart(size=10.0, deviationFactor=0.1,
minSizeFactor=0.1)
```

```
                        partActive.generateMesh()

#---------------------------------------------End of Beams----------------------------------------------------
-
#-----------------------------------------------Columns-----------------------------------------------------
-
#
# Similarly to beams, if the horizontal loads are applied as line loads on the column,
# column_length_factor is calcuated for each column, and is used to calculate the equivalent line loads
#
list_column_length_factor = {}
for i in range(1,bayN+2):
        no_column = i
        for j in range(1,storyN+1):
                no_story = j
                #
                profile_code = columnProfile_code['Story%d'%(j)][i-1]
                top_conn_index = 'Story%d-Connection%d' %(j,i)
                btm_conn_index = 'Story%d-Connection%d'%(j-1,i)
                #
                if profile_code != 'NA':
                        nameStr = 'column-%d-%d'%(i,j)
                        column_index = 'Story%d-column%d'%(j,i)
                        # x-coordinates
                        x_columnEnd = x_connection[i-1]
                        # top end y-coordinates
                        top_conn_code = connection_code['Story%d'%(j)][i-1]
                        if top_conn_code == ('NA','NA'):
                                y_top_columnEnd = y_connection[j]
                        else:
                                y_top_columnEnd                                          =
y_connection[j]+list_y_columnCentre[top_conn_index][1]
                        # btm end y-coordinates
                        if j > 1:
                                btm_conn_code = connection_code['Story%d'%(j-1)][i-1]
                                if btm_conn_code == ('NA','NA'):
                                        y_btm_columnEnd = y_connection[j-1]
                                else:
                                        y_btm_columnEnd            =            y_connection[j-
1]+list_y_columnCentre[btm_conn_index][0]
                        elif j == 1:
                                y_btm_columnEnd = 0
                        # calculate and store column_length_factor
                        column_length = y_top_columnEnd - y_btm_columnEnd
                        column_length_factor = (y_connection[j] - y_connection[j-1])/column_length
                        list_column_length_factor[column_index] = column_length_factor
                        # sketch
```

```
                        sketchActive = myModel.ConstrainedSketch(name='Sketch-' + nameStr,
sheetSize=2000.0)
                        sketchActive.Line(point1=(x_columnEnd,y_top_columnEnd),
point2=(x_columnEnd,y_btm_columnEnd))
                        # part
                        partActive                =                myModel.Part(name='Part-
'+nameStr,dimensionality=THREE_D,type=DEFORMABLE_BODY)
                        partActive.BaseWire(sketch = sketchActive)
                        # material and section
                        materialActive = myModel.Material(name='Material-' + nameStr)
                        youngsModulus = elasticity_library[profile_code]
                        materialActive.Elastic(table = ((youngsModulus,poissonsRatio),))
                        plasticityActive = plasticity_library[profile_code]
                        materialActive.Plastic(table = plasticityActive)
                        geometryActive = profile_library[profile_code]
                        myModel.IProfile(name='Profile-' + nameStr,l=0.5*geometryActive[0],

        h=geometryActive[0],b1=geometryActive[1],b2=geometryActive[1],

        t1=geometryActive[2],t2=geometryActive[2],t3=geometryActive[3])
                        myModel.BeamSection(name='Section-
'+nameStr,integration=DURING_ANALYSIS,poissonRatio=0.0,
                                                        profile='Profile-'    +    nameStr,
material='Material-' + nameStr,
                                                        temperatureVar=LINEAR,
consistentMassMatrix=False)
                        edge_part = partActive.edges
                        region_part = regionToolset.Region(edges=edge_part)
                        partActive.SectionAssignment(region=region_part, sectionName='Section-' +
nameStr,
                                                                        offset=0.0,
offsetType=MIDDLE_SURFACE, offsetField='',

        thicknessAssignment=FROM_SECTION)
                        partActive.assignBeamSectionOrientation(region=region_part,
method=N1_COSINES,

        n1=(0.0, 0.0, -1.0))
                        # instance
                        instance_name = 'Instance-' + nameStr
                        myAssembly.Instance(name = instance_name, part = partActive, dependent =
ON)
                        # mesh
                        element_type=mesh.ElemType(elemCode=B31, elemLibrary=STANDARD)
                       partActive.setElementType(regions=region_part,elemTypes=(element_type, ))
                        partActive.seedPart(size=200.0, deviationFactor=0.1, minSizeFactor=0.1)
                        partActive.generateMesh()
```

```
#----------------------------------------------End of Columns-----------------------------------------------------
--
#----------------------------------------------------------------------------------------------------------------
#--------------------------------------------Interaction module-----------------------------------------------------
-
#----------------------------------------------Constraints---------------------------------------------------------
# Coupling constrain between beam and beam end
# left-hand side of beam
for i in range(1,bayN+1):
	no_beam = i
	for j in range (1,storyN+1):
		no_story = j
		#
		profile_code = beamProfile_code['Story%d'%(j)][i-1]
		left_conn_index = 'Story%d-Connection%d' %(j,i)
		right_conn_index = 'Story%d-Connection%d'%(j,i+1)
		left_conn_code = list(connection_code['Story%d' %(j)][i-1])[1]
		right_conn_code = list(connection_code['Story%d' %(j)][i])[0]
		#
		if profile_code != 'NA':
			if left_conn_code != 'NA' or right_conn_code != 'NA':
				#
				beam_name = 'Instance-beam-%d-%d' %(i,j)
				beamEnd_name = 'Instance-beamEnd-%d-%d' %(i,j)
				#
				if left_conn_code != 'NA':
					constraint_name = 'Constraint-beamEnd-%d-%d-L' %(i,j)
					x_coord = x_connection[i-
1]+list_x_beamEnd[left_conn_index][1]
					y_coord = y_connection[j]
					#
					beamVertex =  myAssembly.instances[beam_name].vertices.\

	findAt(((x_coord,y_coord,0.0),),)
					region_beamVertex =
regionToolset.Region(vertices=beamVertex)
					beamEndVertex =
myAssembly.instances[beamEnd_name].vertices.\

	findAt(((x_coord,y_coord,0.0),),)
					region_beamEndVertex =
regionToolset.Region(vertices=beamEndVertex)
					myModel.Coupling(name = constraint_name,
								controlPoint =
region_beamEndVertex, surface=region_beamVertex,

	influenceRadius=WHOLE_SURFACE, couplingType=KINEMATIC, localCsys=None,
```

```
                                                  u1=ON,   u2=ON,   u3=ON,
ur1=ON, ur2=ON, ur3=ON)
                        #
                        if right_conn_code != 'NA':
                            constraint_name = 'Constraint-beamEnd-%d-%d-R' %(i,j)
                            x_coord                                        =
x_connection[i]+list_x_beamEnd[right_conn_index][0]
                            y_coord = y_connection[j]

                            #
                            beamVertex =  myAssembly.instances[beam_name].vertices.\

    findAt(((x_coord,y_coord,0.0),),)
                            region_beamVertex                              =
regionToolset.Region(vertices=beamVertex)
                            beamEndVertex                                  =
myAssembly.instances[beamEnd_name].vertices.\

    findAt(((x_coord,y_coord,0.0),),)
                            region_beamEndVertex                           =
regionToolset.Region(vertices=beamEndVertex)
                            myModel.Coupling(name = constraint_name,
                                                  controlPoint             =
region_beamEndVertex, surface=region_beamVertex,

    influenceRadius=WHOLE_SURFACE, couplingType=KINEMATIC, localCsys=None,
                                                  u1=ON,   u2=ON,   u3=ON,
ur1=ON, ur2=ON, ur3=ON)

#----------------------------------
# Coupling constrain between beams
# left-hand side of beam
for i in range(1,bayN):
    no_beam = i
    for j in range (1,storyN+1):
        no_story = j
        #
        left_beam_code = beamProfile_code['Story%d'%(j)][i-1]
        right_beam_code = beamProfile_code['Story%d'%(j)][i]
        conn_code = connection_code['Story%d' %(j)][i]
        #
        if left_beam_code != 'NA' and right_beam_code != 'NA' and conn_code == ('NA','NA'):
            #
            left_beam = 'Instance-beam-%d-%d' %(i,j)
            right_beam = 'Instance-beam-%d-%d' %(i+1,j)
            constraint_name = 'Constraint-beam-%d-%d' %(i+1,j)
            x_coord = x_connection[i]
            y_coord = y_connection[j]
```

```
                    #
                    leftVertex =  myAssembly.instances[left_beam].vertices.\
                                                findAt(((x_coord,y_coord,0.0),),)
                    region_leftVertex = regionToolset.Region(vertices=leftVertex)
                    rightVertex = myAssembly.instances[right_beam].vertices.\
                                                findAt(((x_coord,y_coord,0.0),),)
                    region_rightVertex = regionToolset.Region(vertices=rightVertex)
                    myModel.Coupling(name = constraint_name,
                                                    controlPoint    =    region_leftVertex,
surface=region_rightVertex,

        influenceRadius=WHOLE_SURFACE, couplingType=KINEMATIC, localCsys=None,
                                                    u1=ON,  u2=ON,  u3=ON,  ur1=ON,
ur2=ON, ur3=ON)

#-----------------------------------
# Coupling constrain between column centre line and column ends
for i in range(1,bayN+2):
        no_column = i
        for j in range (1,storyN+1):
                no_story = j
                #
                column_code = columnProfile_code['Story%d'%(j)][i-1]
                top_conn_index = 'Story%d-Connection%d' %(j,i)
                btm_conn_index = 'Story%d-Connection%d'%(j-1,i)
                #
                if column_code != 'NA':
                        column_name = 'Instance-column-%d-%d' %(i,j)
                        #
                        # top node constraint
                        conn_code = connection_code['Story%d' %(j)][i-1]
                        if conn_code != ('NA','NA'):
                                conn_name = 'Instance-columnCentre-%d-%d' %(i,j)
                                constraint_name = 'Constraint-columnToConn-%d-%d-T' %(i,j)
                                x_coord = x_connection[i-1]
                                y_coord = y_connection[j] + list_y_columnCentre[top_conn_index][1]

                                columnVertex =  myAssembly.instances[column_name].vertices.\
                                                    findAt(((x_coord,y_coord,0.0),),)
                                region_columnVertex                                          =
regionToolset.Region(vertices=columnVertex)
                                connVertex = myAssembly.instances[conn_name].vertices.\
                                                    findAt(((x_coord,y_coord,0.0),),)
                                region_connVertex = regionToolset.Region(vertices=connVertex)
                                myModel.Coupling(name = constraint_name,
                                                            controlPoint                 =
region_connVertex, surface=region_columnVertex,
```

```
        influenceRadius=WHOLE_SURFACE, couplingType=KINEMATIC, localCsys=None,
                                                    u1=ON,    u2=ON,    u3=ON,
ur1=ON, ur2=ON, ur3=ON)
                    #
                    # btm node constraint
                    if j > 1:
                        conn_code = connection_code['Story%d' %(j-1)][i-1]
                        if conn_code != ('NA','NA'):
                            conn_name = 'Instance-columnCentre-%d-%d' %(i,j-1)
                            constraint_name     =     'Constraint-columnToConn-%d-%d-
B' %(i,j)
                            x_coord = x_connection[i-1]
                            y_coord        =        y_connection[j-1]        +
list_y_columnCentre[btm_conn_index][0]
                            columnVertex                                    =
myAssembly.instances[column_name].vertices.\

        findAt(((x_coord,y_coord,0.0),),)
                            region_columnVertex                             =
regionToolset.Region(vertices=columnVertex)
                            connVertex = myAssembly.instances[conn_name].vertices.\

        findAt(((x_coord,y_coord,0.0),),)
                            region_connVertex                               =
regionToolset.Region(vertices=connVertex)
                            myModel.Coupling(name = constraint_name,
                                                        controlPoint        =
region_connVertex, surface=region_columnVertex,

        influenceRadius=WHOLE_SURFACE, couplingType=KINEMATIC, localCsys=None,
                                                        u1=ON,      u2=ON,
u3=ON, ur1=ON, ur2=ON, ur3=ON)

#----------------------------------
# Coupling constrain between columns
# left-hand side of beam
for i in range(1,bayN+2):
        no_beam = i
        for j in range (1,storyN):
                no_story = j
                #
                btm_column_code = columnProfile_code['Story%d'%(j)][i-1]
                top_column_code = columnProfile_code['Story%d'%(j+1)][i-1]
                conn_code = connection_code['Story%d' %(j)][i-1]
                #
                if  btm_column_code  !=  'NA'  and  top_column_code  !=  'NA'  and  conn_code  ==
('NA','NA'):
```

```
                    #
                    btm_column = 'Instance-column-%d-%d' %(i,j)
                    top_column = 'Instance-column-%d-%d' %(i,j+1)
                    constraint_name = 'Constraint-column-%d-%d' %(i,j)
                    x_coord = x_connection[i-1]
                    y_coord = y_connection[j]
                    #
                    btmVertex =  myAssembly.instances[btm_column].vertices.\
                                        findAt(((x_coord,y_coord,0.0),),)
                    region_btmVertex = regionToolset.Region(vertices=btmVertex)
                    topVertex = myAssembly.instances[top_column].vertices.\
                                        findAt(((x_coord,y_coord,0.0),),)
                    region_topVertex = regionToolset.Region(vertices=topVertex)
                    myModel.Coupling(name = constraint_name,
                                                controlPoint = region_btmVertex,
surface=region_topVertex,

        influenceRadius=WHOLE_SURFACE, couplingType=KINEMATIC, localCsys=None,
                                                u1=ON, u2=ON, u3=ON, ur1=ON,
ur2=ON, ur3=ON)

#-----------------------------------------------End of Constraints--------------------------------------------------
-
#-------------------------------------------------Connectors---------------------------------------------------------
# connectors for the right-hand side connections
refPoint_id_overall = []
for i in range(1,bayN+2):
        no_column = i
        for j in range(1,storyN+1):
                no_story = j
                #
                # connection code
                conn_code = connection_code['Story%d' %(j)][i-1][1]
                #
                if conn_code != 'NA':
                        conn_index = 'Story%d-Connection%d' %(j,i)
                        columnCentre_name = "Instance-columnCentre-%d-%d" % (i,j)
                        beamEnd_name = "Instance-beamEnd-%d-%d" % (i,j)
                        x_columnCentre = x_connection[i-1]
                        x_beamEnd_local = list_x_beamEnd[conn_index][1]
                        x_beamEnd = x_connection[i-1] + x_beamEnd_local
                        conn_prop = connection_library[conn_code]
                        springRow_keys = conn_prop.keys()
                        #-------------------------------------
                        # Cartesian type connector to transfer shear force from beam to column, at
beam centre line
                        # Create wires
                        y_springRow = y_connection[j]
```

```
                        vertex_columnCentre                                    =
myAssembly.instances[columnCentre_name].vertices.\

                findAt(((x_columnCentre,y_springRow,0.0),),)
                        vertex_beamEnd = myAssembly.instances[beamEnd_name].vertices.\

                findAt(((x_beamEnd,y_springRow,0.0),),)
                        tuple_of_points = (vertex_columnCentre[0],vertex_beamEnd[0])
                        myAssembly.WirePolyLine(points=(tuple_of_points,),mergeType=IMPRINT,
meshable=OFF)
                        #
                        # Create connector section
                        connSect_name = 'ConnSect-%d-%d-R-beamCentre' %(i,j)
                        set_connector = "Conn-%d-%d-R-beamCentre" % (i,j)
                        myModel.ConnectorSection(name          =          connSect_name,
translationalType=CARTESIAN)
                        sectionConn = myModel.sections[connSect_name]
                        prop2                                                  =
connectorBehavior.ConnectorElasticity(components=(2, ),behavior=RIGID)
                        sectionConn.setValues(behaviorOptions =(prop2,))
                        sectionConn.behaviorOptions[0].ConnectorOptions( )
                        # Assign the wire features/connectors to a set
                        x_coord = x_columnCentre
                        y_coord = y_springRow
                        edges_for_setConnector                                 =
myAssembly.edges.findAt(((x_coord,y_coord,0.0),),)
                        # assign connector section to connector wires
                        conn_region      =      myAssembly.Set(edges=edges_for_setConnector,
name=set_connector)

        myAssembly.SectionAssignment(sectionName=connSect_name,region=conn_region)
                        #--------------------------------------
                        # Connectors for bolt rows
                        # For each spring row
                        for n in range(len(springRow_keys)):
                                springRow_code = springRow_keys[n]
                                y_dis = conn_prop[springRow_code]['y_dis']
                                y_springRow = y_connection[j] + y_dis
                                componentN = conn_prop[springRow_code]['no_component']
                                if componentN > 1:
                                        rp_Logic = True
                                # if a bolt row contains multiple components, create reference points
in order to
                                # create wires representing the components
                                refPoint_id = [] # record the id of the ref points
                                dis_refPoint = x_beamEnd_local/componentN    # distance between
ref points
                                #
```

```
                                        # Create reference points, and, find vertices on the vertical rigid bar
and beam end
                                        # put the vertices and ref points in a list (list_vertices) in the order of
                                        # [vertex_columnCentre, refPoints, vertex_beamEnd]
                                        list_vertices = []
                                        # record the x-coordinate of a point on each component, don't use the
points on the
                                        # vertical rigid bar because the points may also correspond to the shear
panel component
                                        list_x_coord_for_edge = []
                                        # find and record the vertex on the vertical rigid bar
                                        vertex_columnCentre                                        =
myAssembly.instances[columnCentre_name].vertices.\

                    findAt(((x_columnCentre,y_springRow,0.0),),)
                                        list_vertices.append(vertex_columnCentre[0])
                                        # create and record the ref points
                                        # note: x_coord_for_edge: x_columnCentre + (h-0.5) * dis_refPoint
                                        for h in range(1,componentN):
                                                  x_refPoint = x_columnCentre + h * dis_refPoint
                                                  refPoint   =   myAssembly.ReferencePoint(point=(x_refPoint,
y_springRow, 0.0))
                                                  refPoint_id.append(refPoint.id)
                                                  refPoint_id_overall.append(refPoint.id)

          list_vertices.append(myAssembly.referencePoints[refPoint_id[-1]])
                                                  #
                                                  x_coord_for_edge =  x_columnCentre + (h-0.5) * dis_refPoint
                                                  list_x_coord_for_edge.append(x_coord_for_edge)
                                        # find and record the vertex on the beam end
                                        vertex_beamEnd = myAssembly.instances[beamEnd_name].vertices.\

                    findAt(((x_beamEnd,y_springRow,0.0),),)
                                        list_vertices.append(vertex_beamEnd[0])
                                        list_x_coord_for_edge.append(x_beamEnd - 0.5* dis_refPoint)
                                        #
                                        # For each component
                                        for g in range(1,componentN+1):
                                                  # Create wires
                                                  tuple_of_points = (list_vertices[g-1],list_vertices[g])

          myAssembly.WirePolyLine(points=(tuple_of_points,),mergeType=IMPRINT,
meshable=OFF)
                                                  #
                                                  # Create connector section
                                                  component_index = 'component-%d' %(g)
                                                  connSect_name     =     'ConnSect-%d-%d-R-'     %(i,j)     +
springRow_code + '-%d' %(g)
```

```
                                        set_connector = "Conn-%d-%d-R-" % (i,j) + springRow_code
+ '-%d' %(g)
                                        x_behaviour                                          =
conn_prop[springRow_code][component_index]
                                        # components in the bolt row has CARTESIAN connector
type,
                                        myModel.ConnectorSection(name     =     connSect_name,
translationalType=AXIAL)
                                        sectionConn = myModel.sections[connSect_name]
                                        prop1                                                =
connectorBehavior.ConnectorElasticity(components=(1, ),

                                        behavior=NONLINEAR,table=x_behaviour)
                                        damping1                                             =
connectorBehavior.ConnectorDamping(components=(1, ), table=((100.0, ), ))
                                       sectionConn.setValues(behaviorOptions =(prop1,damping1,))
                                        sectionConn.behaviorOptions[0].ConnectorOptions( )
                                        sectionConn.behaviorOptions[1].ConnectorOptions( )
                                        #
                                        # Assign the wire features/connectors to a set
                                        x_coord = list_x_coord_for_edge[g-1]
                                        y_coord = y_springRow
                                        if y_coord != 0:
                                                edges_for_setConnector                       =
myAssembly.edges.findAt(((x_coord,y_coord,0.0),),)
                                        elif y_coord == 0:
                                                torl = 1.0
                                                x_Min = x_coord - 0.5*dis_refPoint - torl
                                                x_Max = x_coord + 0.5*dis_refPoint + torl
                                                y_Min = y_coord - torl
                                                y_Max = y_coord + torl
                                                z_Min = 0.0 - torl
                                                z_Max = 0.0 + torl

                                                edges_for_setConnector                       =
myAssembly.edges.getByBoundingBox(x_Min,y_Min,z_Min,x_Max,y_Max,z_Max)
                                        # assign connector section to connector wires
                                        conn_region                                          =
myAssembly.Set(edges=edges_for_setConnector, name=set_connector)

        myAssembly.SectionAssignment(sectionName=connSect_name,region=conn_region)

#
#----------------------------------
# Connectors for the left-hand side bolt rows
for i in range(1,bayN+2):
        no_column = i
        for j in range(1,storyN+1):
```

```
        no_story = j
        #
        # connection code
        conn_code = connection_code['Story%d' %(j)][i-1][0]
        #
        if conn_code != 'NA':
            conn_index = 'Story%d-Connection%d' %(j,i)
            columnCentre_name = "Instance-columnCentre-%d-%d" % (i,j)
            beamEnd_name = "Instance-beamEnd-%d-%d" % (i-1,j)
            x_columnCentre = x_connection[i-1]
            x_beamEnd_local = list_x_beamEnd[conn_index][0]
            x_beamEnd = x_connection[i-1] + x_beamEnd_local
            conn_prop = connection_library[conn_code]
            springRow_keys = conn_prop.keys()
            #-------------------------------------
            # Cartesian type connector to transfer shear force from beam to column, at
beam centre line
            # Create wires
            y_springRow = y_connection[j]
            vertex_columnCentre                                                    =
myAssembly.instances[columnCentre_name].vertices.\

        findAt(((x_columnCentre,y_springRow,0.0),),)
            vertex_beamEnd = myAssembly.instances[beamEnd_name].vertices.\

        findAt(((x_beamEnd,y_springRow,0.0),),)
            tuple_of_points = (vertex_columnCentre[0],vertex_beamEnd[0])
            myAssembly.WirePolyLine(points=(tuple_of_points,),mergeType=IMPRINT,
meshable=OFF)
            #
            # Create connector section
            connSect_name = 'ConnSect-%d-%d-L-beamCentre' %(i,j)
            set_connector = "Conn-%d-%d-L-beamCentre" % (i,j)
            myModel.ConnectorSection(name          =          connSect_name,
translationalType=CARTESIAN)
            sectionConn = myModel.sections[connSect_name]
            prop2                                                                  =
connectorBehavior.ConnectorElasticity(components=(2, ),behavior=RIGID)
            sectionConn.setValues(behaviorOptions =(prop2,))
            sectionConn.behaviorOptions[0].ConnectorOptions( )
            # Assign the wire features/connectors to a set
            x_coord = x_beamEnd
            y_coord = y_springRow
            edges_for_setConnector                                                 =
myAssembly.edges.findAt(((x_coord,y_coord,0.0),),)
            # assign connector section to connector wires
            conn_region      =      myAssembly.Set(edges=edges_for_setConnector,
name=set_connector)
```

```
myAssembly.SectionAssignment(sectionName=connSect_name,region=conn_region)
                    #-------------------------------------
                    # Connectors for bolt rows
                    # For each spring row
                    for n in range(len(springRow_keys)):
                              springRow_code = springRow_keys[n]
                              y_dis = conn_prop[springRow_code]['y_dis']
                              y_springRow = y_connection[j] + y_dis
                              componentN = conn_prop[springRow_code]['no_component']
                              if componentN > 1:
                                        rp_Logic = True
                              # if a bolt row contains multiple components, create reference points
in order to
                              # create wires representing the components
                              refPoint_id = [] # record the id of the ref points
                              dis_refPoint = abs(x_beamEnd_local)/componentN      #    distance
between ref points
                              #
                              # Create reference points, and, find vertices on the vertical rigid bar
and beam end
                              # put the vertices and ref points in a list (list_vertices) in the order of
                              # [vertex_beamEnd, refPoints, vertex_vtcl]
                              list_vertices = []
                              # record the x-coordinate of a point on each component, don't use the
points on the
                              # vertical rigid bar because the points may also correspond to the shear
panel component
                              list_x_coord_for_edge = []
                              # find and record the vertex on the beam end
                              vertex_beamEnd = myAssembly.instances[beamEnd_name].vertices.\

          findAt((((x_beamEnd,y_springRow,0.0),),)
                              list_vertices.append(vertex_beamEnd[0])
                              list_x_coord_for_edge.append(x_beamEnd + 0.5 * dis_refPoint)

                              # create and record the ref points
                              # note: x_coord_for_edge: x_beamEnd + (h+0.5) * dis_refPoint
                              for h in range(1,componentN):
                                        x_refPoint = x_beamEnd + h * dis_refPoint
                                        refPoint   =   myAssembly.ReferencePoint(point=(x_refPoint,
y_springRow, 0.0))
                                        refPoint_id.append(refPoint.id)
                                        refPoint_id_overall.append(refPoint.id)

     list_vertices.append(myAssembly.referencePoints[refPoint_id[-1]])
                                        #
                                        x_coord_for_edge =  x_beamEnd + (h+0.5) * dis_refPoint
```

```
                                        list_x_coord_for_edge.append(x_coord_for_edge)
                                # find and record the vertex on the vertical rigid bar
                                vertex_columnCentre                                   =
myAssembly.instances[columnCentre_name].vertices.\

                findAt(((x_columnCentre,y_springRow,0.0),),)
                                list_vertices.append(vertex_columnCentre[0])
                                #
                                # For each component
                                for g in range(1,componentN+1):
                                        # Create wires
                                        tuple_of_points = (list_vertices[g-1],list_vertices[g])

        myAssembly.WirePolyLine(points=(tuple_of_points,),mergeType=IMPRINT,
meshable=OFF)
                                        #
                                        # Create connector section
                                        component_index = 'component-%d' %(g)
                                        connSect_name     =     'ConnSect-%d-%d-L-'     %(i,j)     +
springRow_code + '-%d' %(g)
                                        set_connector = "Conn-%d-%d-L-" % (i,j) + springRow_code
+ '-%d' %(g)
                                        x_behaviour                                           =
conn_prop[springRow_code][component_index]
                                        # components in the bolt row has AXIAL connector type,
unless it is at the beam centre
                                        myModel.ConnectorSection(name      =      connSect_name,
translationalType=AXIAL)
                                        sectionConn = myModel.sections[connSect_name]
                                        prop1                                                 =
connectorBehavior.ConnectorElasticity(components=(1, ),

                                                behavior=NONLINEAR,table=x_behaviour)
                                        damping1                                              =
connectorBehavior.ConnectorDamping(components=(1, ), table=((100.0, ), ))
                                        sectionConn.setValues(behaviorOptions =(prop1,damping1,))
                                        sectionConn.behaviorOptions[0].ConnectorOptions( )
                                        sectionConn.behaviorOptions[1].ConnectorOptions( )
                                        #
                                        # Assign the wire features/connectors to a set
                                        x_coord = list_x_coord_for_edge[g-1]
                                        y_coord = y_springRow
                                        if y_coord != 0:
                                                edges_for_setConnector                        =
myAssembly.edges.findAt(((x_coord,y_coord,0.0),),)
                                        elif y_coord == 0:
                                                torl = 1.0
                                                x_Min = x_coord - 0.5*dis_refPoint - torl
```

```
                                                x_Max = x_coord + 0.5*dis_refPoint + torl
                                                y_Min = y_coord - torl
                                                y_Max = y_coord + torl
                                                z_Min = 0.0 - torl
                                                z_Max = 0.0 + torl

                                                edges_for_setConnector                         =
myAssembly.edges.getByBoundingBox(x_Min,y_Min,z_Min,x_Max,y_Max,z_Max)
                                        # assign connector section to connector wires
                                        conn_region                                    =
myAssembly.Set(edges=edges_for_setConnector, name=set_connector)

        myAssembly.SectionAssignment(sectionName=connSect_name,region=conn_region)

#----------------------------------
# set of connectors for output
set_connector_name = 'Set-ConnOutput'
myAssembly.Set(edges = myAssembly.edges, name = set_connector_name)
#
#--------------------------------------------End of Connectors---------------------------------------------------
--
#-----------------------------------------End of Interaction module----------------------------------------------
---
#------------------------------------------------------------------------------------------------------------
#-----------------------------------------------Step  module-------------------------------------------------------
-
# Normal static analysis, only force controlled loading mode
# two analysis modes, static and riks
if analysisMode == "static":
        myModel.StaticStep(name='Step-loading', previous='Initial',
                                                        maxNumInc=1000,     initialInc=0.1,
maxInc=0.2, nlgeom=ON)
elif analysisMode == "riks":
        myModel.StaticRiksStep(name='Step-loading', previous='Initial',
                                                                maxNumInc=1000,
initialArcInc=0.1, nlgeom=ON)
else:
        raise ValueError ("Error,analysis type is missing or wrong")

#
# Filed output requests
# define output for connectors
myModel.fieldOutputRequests['F-Output-1'].setValues(variables=('S', 'PE', 'PEEQ', 'PEMAG',
        'LE', 'U', 'RF', 'CF', 'SF', 'CSTRESS', 'CDISP'))
Set_connOutput = myAssembly.sets['Set-ConnOutput']
myModel.FieldOutputRequest(name='F-Output-Conn', createStepName='Step-loading',
                                                variables=('CTF', 'CEF', 'CU', 'CUE', 'CUP'),
```

```
                                                region=Set_connOutput,
sectionPoints=DEFAULT, rebar=EXCLUDE)

# History output request
# leave at defaults
#
#----------------------------------------------End of Step module--------------------------------------------------
--
#---------------------------------------------------------------------------------------------------------------
#----------------------------------------------Load  module------------------------------------------------------
--
#--------------------------------------------------Load -----------------------------------------------------------
# Load applied in forceLoading Mode
# Vertial load on beams
if vLoad_type == 'line_load':
	for i in range(1,bayN+1):
		for j in range (1,storyN+1):
			story_code = 'Story%d'%(j)
			v_load = vLoad[story_code][i-1]
			if v_load != 'NA':
				#
				beam_name = 'Instance-beam-%d-%d'%(i,j)
				load_name = 'Load-vLoad-line-%d-%d' %(i,j)
				edge_for_load = myAssembly.instances[beam_name].edges
				region_for_load = regionToolset.Region(edges=edge_for_load)
				#
				# the line loads are applied on the beam, which is shorter than the bay
span
				# due to the connection width, so have to use the equivalent line load
				beam_index = 'Story%d-beam%d'%(j,i)
				beam_length_factor = list_beam_length_factor[beam_index]
				myModel.LineLoad(name = load_name, createStepName = 'Step-
loading',
                                                    region=region_for_load, comp2 = -
1*beam_length_factor*float(v_load))
elif vLoad_type == 'point_load':
	for i in range(1,bayN+2):
		for j in range (1,storyN+1):
			story_code = 'Story%d'%(j)
			storyTop_code = 'Story%d'%(j+1)
			v_load = vLoad[story_code][i-1]
			if v_load != 'NA':
				load_name = 'Load-vLoad-point-%d-%d' %(i,j)
				conn_code = connection_code['Story%d' %(j)][i-1]
				x_conn = x_connection[i-1]
				y_conn = y_connection[j]
				beamLeft_code = beamProfile_code_plus[story_code][i-1]
				beamRight_code = beamProfile_code_plus[story_code][i]
```

```
                        columnBtm_code = columnProfile_code_plus[story_code][i-1]
                        columnTop_code = columnProfile_code_plus[storyTop_code][i-1]
                        # if there is a connection (i.e. a column), the beam is applied on the
column centre line
                        if conn_code != ('NA','NA'):
                            load_Instance_nameStr = 'Instance-
columnCentre-%d-%d' %(i,j)
                        # if there is no connection, but two beams
                        elif (conn_code == ('NA','NA')) and (beamLeft_code != 'NA') and
(beamRight_code != 'NA'):
                            load_Instance_nameStr = 'Instance-beam-%d-%d' %(i,j)
                        # if there is no connection, but two columns
                        elif (conn_code == ('NA','NA')) and (columnBtm_code != 'NA') and
(columnTop_code != 'NA'):
                            load_Instance_nameStr = 'Instance-column-%d-%d' %(i,j)
                        vertex_load =
myAssembly.instances[load_Instance_nameStr].vertices.findAt(((x_conn,y_conn,0.0),),)
                        region_vertex = regionToolset.Region(vertices=vertex_load)
                        myModel.ConcentratedForce(name =
load_name,createStepName='Step-loading',

    region=region_vertex, cf2=-1000*float(v_load), distributionType=UNIFORM,
                                                field='',
localCsys=None)

#----------------------------------
# Horizontal load on the columns
if hLoad_type != 'NA':
    for i in range(1,bayN+2):
        for j in range (1,storyN+1):
            h_load = hLoad['Story%d'%(j)][i-1]
            story_code = 'Story%d'%(j)
            if h_load != 'NA':
                load_name = 'Load-hLoad-line-%d-%d' %(i,j)
                print load_name
                if hLoad_type == 'line_load':
                    column_nameStr = 'Instance-column-%d-%d'%(i,j)
                    edge_for_load =
myAssembly.instances[column_nameStr].edges
                    region_for_load =
regionToolset.Region(edges=edge_for_load)
                    #
                    # the line loads are applied on the column, which is shorter
than the story height
                    # due to the connection height, so have to use the equivalent
line load
                    column_index = 'Story%d-column%d'%(j,i)
```

```
                                        column_length_factor                                    =
list_column_length_factor[column_index]
                                        #
                                        myModel.LineLoad(name = load_name, createStepName =
'Step-loading',
                                                                    region=region_for_load,
comp1 = column_length_factor*float(h_load))
                                elif hLoad_type == 'point_load':
                                        conn_code = connection_code['Story%d' %(j)][i-1]
                                        x_conn = x_connection[i-1]
                                        y_conn = y_connection[j]
                                        if conn_code != ('NA','NA'):
                                                load_Instance_nameStr          =          'Instance-
columnCentre-%d-%d' %(i,j)
                                        elif conn_code == ('NA','NA'):
                                                load_Instance_nameStr          =          'Instance-
column-%d-%d' %(i,j)
                                        vertex_load                                             =
myAssembly.instances[load_Instance_nameStr].vertices.findAt(((x_conn,y_conn,0.0),),)
                                        region_vertex = regionToolset.Region(vertices=vertex_load)
                                        myModel.ConcentratedForce(name                          =
load_name,createStepName='Step-loading',

        region=region_vertex, cf1=1000*float(h_load), distributionType=UNIFORM,
                                                                            field='',
localCsys=None)

#
#-----------------------------------------------End of Load---------------------------------------------------------
#-------------------------------------------Boundary  condition--------------------------------------------------
---
# Boundary condition at the column base
for i in range(1,bayN+2):
        bc_column =  bcColumn['Story1'][i-1]
        column_code = columnProfile_code['Story1'][i-1]
        #
        if column_code != 'NA':
                column_name = 'Instance-column-%d-1' % (i)
                bc_name = 'BC-columnBase-%d' % (i)
                #
                x_bc = x_connection[i-1]
                y_bc = y_connection[0]
               bcVertex =  myAssembly.instances[column_name].vertices.findAt(((x_bc,y_bc,0.0),),)
                region_bcVertex = regionToolset.Region(vertices=bcVertex)
                if bc_column == 'rigid':
                        myModel.DisplacementBC(name = bc_name, createStepName='Initial',
                                                                  region=region_bcVertex,u1=SET,
u2=SET, u3=SET, ur1=SET,
```

```
                                                                        ur2=SET,                ur3=SET,
amplitude=UNSET, distributionType=UNIFORM,
                                                                        fieldName='',localCsys=None)
                elif bc_column == 'pin':
                        myModel.DisplacementBC(name = bc_name, createStepName='Initial',
                                                                        region=region_bcVertex,u1=SET,
u2=SET, u3=SET, ur1=SET,
                                                                        ur2=SET,              ur3=UNSET,
amplitude=UNSET, distributionType=UNIFORM,
                                                                        fieldName='',localCsys=None)

# For planar frame, provide out-of-plane boundary conditions on the beams
# on connections - columnCentre
for i in range(1,bayN+2):
        for j in range(1,storyN+1):
                #
                # connection code
                conn_code = list(connection_code['Story%d' %(j)][i-1])
                #
                if conn_code != ['NA','NA']:
                        # vtclRigid
                        instance_name = 'Instance-columnCentre-%d-%d' %(i,j)
                        bc_name = 'BC-oop-columnCentre-%d-%d' %(i,j)
                        vertexActive = myAssembly.instances[instance_name].vertices
                        region_vertices = regionToolset.Region(vertices=vertexActive)
                        myModel.DisplacementBC(name=bc_name, createStepName='Initial',
                                                                                region=region_vertices,
u1=UNSET, u2=UNSET, u3=SET,
                                                                                ur1=SET,          ur2=SET,
ur3=UNSET, amplitude=UNSET,

        distributionType=UNIFORM, fieldName='', localCsys=None)

# on ref points
if rp_Logic:
        all_refPoint = []
        for i in range(len(refPoint_id_overall)):
                id_refPoint = refPoint_id_overall[i]
                bc_name = 'BC-oop-refPoint-%d' %(id_refPoint)
                all_refPoint.append(myAssembly.referencePoints[id_refPoint])
        region_vertices = regionToolset.Region(referencePoints=all_refPoint)
        myModel.DisplacementBC(name=bc_name, createStepName='Initial',
                                                                region=region_vertices,          u1=UNSET,
u2=UNSET, u3=SET,
                                                                ur1=SET,         ur2=SET,         ur3=SET,
amplitude=UNSET,
                                                                distributionType=UNIFORM,   fieldName='',
localCsys=None)
```

```
# on beams
for i in range(1,bayN+1):
    for j in range(1,storyN+1):
        #
        profile_code = beamProfile_code['Story%d'%(j)][i-1]
        if profile_code != 'NA':
            #
            # Beam
            instance_name = 'Instance-beam-%d-%d'%(i,j)
            bc_name = 'BC-oop-beam-%d-%d' %(i,j)
            vertexActive = myAssembly.instances[instance_name].vertices
            region_vertices = regionToolset.Region(vertices=vertexActive)
            myModel.DisplacementBC(name=bc_name, createStepName='Initial',
                                                            region=region_vertices,
u1=UNSET, u2=UNSET, u3=SET,
                                                            ur1=SET,        ur2=SET,
ur3=UNSET, amplitude=UNSET,

    distributionType=UNIFORM, fieldName='', localCsys=None)
            # Beam end
            left_conn_code = connection_code['Story%d'%(j)][i-1][1]
            right_conn_code = connection_code['Story%d'%(j)][i][0]
            if left_conn_code != 'NA' or right_conn_code != 'NA':
                instance_name = 'Instance-beamEnd-%d-%d'%(i,j)
                bc_name = 'BC-oop-beamEnd-%d-%d' %(i,j)
                vertexActive = myAssembly.instances[instance_name].vertices
                region_vertices = regionToolset.Region(vertices=vertexActive)
                myModel.DisplacementBC(name=bc_name,
createStepName='Initial',

    region=region_vertices, u1=UNSET, u2=UNSET, u3=SET,
                                                                ur1=SET,  ur2=SET,
ur3=UNSET, amplitude=UNSET,

    distributionType=UNIFORM, fieldName='', localCsys=None)

# on columns
for i in range(1,bayN+2):
    for j in range(1,storyN+1):
        #
        profile_code = columnProfile_code['Story%d'%(j)][i-1]
        if profile_code != 'NA':
            instance_name = 'Instance-column-%d-%d'%(i,j)
            bc_name = 'BC-oop-column-%d-%d' %(i,j)
            vertexActive = myAssembly.instances[instance_name].vertices
            region_vertices = regionToolset.Region(vertices=vertexActive)
            myModel.DisplacementBC(name=bc_name, createStepName='Initial',
```

```
                                                            region=region_vertices,
u1=UNSET, u2=UNSET, u3=SET,
                                                            ur1=SET,            ur2=SET,
ur3=UNSET, amplitude=UNSET,

        distributionType=UNIFORM, fieldName='', localCsys=None)

#-------------------------------------------End of Boundary condition--------------------------------------------
----
#---------------------------------------------End of Load module-----------------------------------------------
---
#-------------------------------------------------------------------------------------------------------------
#------------------------------------------------Job  module---------------------------------------------------
-
# Create the job
myAssembly.regenerate()
job_name = "Job-" + model_name
mdb.Job(name=job_name, model='Model-' +model_name, description='', type=ANALYSIS,
atTime=None, waitMinutes=0,
                waitHours=0, queue=None, memory=90, memoryUnits=PERCENTAGE,
getMemoryFromAnalysis=True,
                explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE, echoPrint=OFF,
modelPrint=OFF,
                contactPrint=OFF, historyPrint=OFF, userSubroutine='', scratch='',
resultsFormat=ODB,
                multiprocessingMode=DEFAULT,                    numCpus=cpuN,
numDomains=cpuN,numGPUs=0)
#
# Run the job
if runJob:
        mdb.jobs[job_name].submit(consistencyChecking=OFF)

#---------------------------------------------End of Job module------------------------------------------------
--
#-------------------------------------------------------------------------------------------------------------
#--------------------------------------------Visulazation module----------------------------------------------
#
#
#
#
#
#-------------------------------------------End of Visulazation module-------------------------------------------
----
#-------------------------------------------------------------------------------------------------------------
```

11. Appendix C

Input information to create a finite-element model for an example two-bay two-storey irregular frame.

```
#--------------------------------------------------------------------------------
# 1. FRAME LAYOUT AND GEOMETRY
Bay number:     2
Story number:   2
#               bay 1    bay 2
Bay span:       L1       L2
#               story 1  story 2
Story height:   H1       H2
#--------------------------------------------------------------------------------
# 2. BEAMS AND COLUMNS
# Profiles and material
#          height  width  t_flange  t_web   stress-strain
Profile1:  h1      w1     t_f1      t_w1    pointwise σ-ε
Profile2:  h2      w2     t_f2      t_w2    pointwise σ-ε
Profile3:  h3      w3     t_f3      t_w3    pointwise σ-ε
Profile4:  h4      w4     t_f4      t_w4    pointwise σ-ε
#Profile for beams and columns
#         beam 1     beam 2
Story1:   Profile1   Profile1
Story2:   Profile2   NA
#         column 1   column 2   column 3
Story1:   Profile3   NA         Profile4
Story2:   Profile3   Profile3   NA
#------------------------------------------------------------------------------------------------------
# 3. CONNECTIONS
# Connection properties
#         row         height*  no of cmpt   F-Δ of each component      (*height is distance from beam axis)
Conn1:  top flng    h_tf     2          cmpt-1 F-Δ   cmpt-2 F-Δ
Conn1:  bolt row1   h_br1    4          cmpt-1 F-Δ   cmpt-2 F-Δ   cmpt-3 F-Δ   cmpt-4 F-Δ
Conn1:  bolt row2   h_br2    4          cmpt-1 F-Δ   cmpt-2 F-Δ   cmpt-3 F-Δ   cmpt-4 F-Δ
Conn1:  btm flng    h_bf     2          cmpt-1 F-Δ   cmpt-2 F-Δ
#
Conn2:  boltrow1    h_br1    4          cmpt-1 F-Δ   cmpt-2 F-Δ   cmpt-3 F-Δ   cmpt-4 F-Δ
Conn2:  top flng    h_tf     3          cmpt-1 F-Δ   cmpt-2 F-Δ   cmpt-3 F-Δ
Conn2:  bolt row2   h_br2    4          cmpt-1 F-Δ   cmpt-2 F-Δ   cmpt-3 F-Δ   cmpt-4 F-Δ
Conn2:  bolt row3   h_br3    4          cmpt-1 F-Δ   cmpt-2 F-Δ   cmpt-3 F-Δ   cmpt-4 F-Δ
Conn2:  btm flng    h_bf     3          cmpt-1 F-Δ   cmpt-2 F-Δ   cmpt-3 F-Δ
#
Conn3:  bolt row1   h_br1    3          cmpt-1 F-Δ   cmpt-2 F-Δ   cmpt-3 F-Δ
Conn3:  top flng    h_tf     2          cmpt-1 F-Δ   cmpt-2 F-Δ
Conn3:  btm flng    h_bf     2          cmpt-1 F-Δ   cmpt-2 F-Δ
Conn3:  bolt_row2   h_br2    3          cmpt-1 F-Δ   cmpt-2 F-Δ   cmpt-3 F-Δ
# Configuration for each connection
#         column 1       column 2        column 3
Story1:  (NA, Conn1)    (Conn1, Conn1)  (Conn2, NA)
Story2:  (NA, Conn3)    (Conn3, NA)     (NA, NA)
#------------------------------------------------------------------------------------------------------
# 4. LOAD AND BOUNDARY CONDITION
Vertical load type: line_load
#         bay 1   bay 2
Story1:  v1      v1
```

```
Story2:  v2         v2
#
Horizontal load type:  point_load
#          column 1    column 2    column 3
Story1:  h1            NA           NA
Story2:  h1            NA           NA
# Boundary condition
#          column 1    column 2    column 3
Story1:  rigid         NA           rigid
#----------------------------------------------------------------------------------------------------------
```

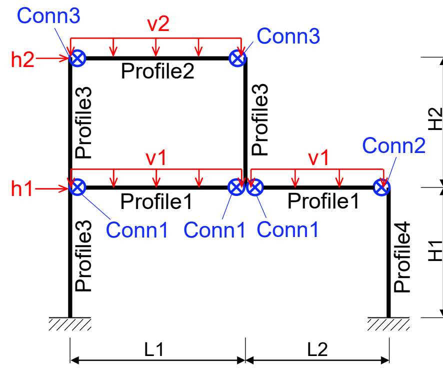


Fig. C-1 Example of steel frame analysis input file.