

**DESIGN OF PARTIALLY RESTRAINED STEEL FRAMES
USING ADVANCED ANALYSIS AND
AN OBJECT-ORIENTED EVOLUTIONARY ALGORITHM**

by

Daniel C. Schinler

A Thesis submitted to the Faculty of the Graduate School,
Marquette University, in Partial Fulfillment of the
Requirements for the Degree of Master of Science

Milwaukee, Wisconsin

Preface

The design of steel frames often involves an optimization process, in which the design evolves from an initial to a final configuration. The goal of most optimal steel design problems is to minimize the cost while satisfying performance and construction criteria. Traditionally, the design problem has been solved by trial-and-error dictated by design specifications and guided by the experience and intuition of the designer. However, researchers are continually developing analysis and optimization tools to assist engineers in the sometimes laborious design process and to foster creativity in arriving at “the optimal” design. Over the past decade, the innovation of these design tools has escalated with the advent of high-speed computer processors.

The focus of this thesis is the development of an optimization algorithm to design fully-restrained (FR) and partially-restrained (PR) steel frames. Upon review of the optimization techniques available in the literature, an evolutionary algorithm (EA) was selected to stochastically guide the algorithm through the solution space of available designs and arrive at an evolved frame. Furthermore, the EA implements an object-oriented heuristic tree representation of the design variables (*i.e.* member sizes and connection stiffness).

A method of advanced analysis is used to assess the adequacy of the steel frames in lieu of design specification and code requirements. The advanced analysis based design uses an inelastic analysis capable of capturing the material and geometric nonlinear behavior of the frame members and incorporates a connection model to capture the nonlinear behavior associated with the connections.

The EA is specific to the design of unbraced FR and PR steel frames with known geometry, material properties, and loading. Three frames selected from the literature are designed using the EA. The performance of the EA is assessed using convergence trajectories, load deformation responses of the evolved frames, and results from previous researchers.

Conclusions are made as to the performance of the evolutionary algorithm and the structural behavior of the evolved frames. Suggestions for future applications of the algorithm are provided and recommendations to improve the performance of the algorithm are outlined as well.

Acknowledgements

I would like to express my sincere appreciation to my thesis director, Dr. Christopher M. Foley for his guidance and support throughout this research effort. I consider myself fortunate to have had a mentor with the strong work ethic and unyielding patience of Dr. Foley.

I would like to thank Professor Sriramulu Vinnakota for his expertise in reviewing this thesis and his instrumental role in my return to graduate school. I am also grateful for the contributions of Professor Stephen M. Heinrich both as member of my thesis committee and as an exceptional teacher.

I would also like to acknowledge Mark S. Voss for his initial contributions related to genetic algorithms and the object-oriented programming language, Python.

The National Science Foundation supported this research under Grant Number CMS-983216.

Table of Contents

1	Introduction	1
1.1	Objective and Scope	1
1.2	Background and Literature Review	2
1.2.1	Structural Optimization	2
	<i>Gradient-based Techniques</i>	
	<i>Direct (Stochastic) Search Techniques</i>	
1.2.2	Advanced Analysis	8
1.2.3	Object-Oriented Programming	10
1.3	Overview of Contents	15
2	Methods of Optimization	16
2.1	Gradient-Based Techniques	17
2.1.1	Nonlinear Programming Problem	19
2.1.2	Optimality Criteria	20
2.1.3	Bound and Branch Method	22
2.2	Direct (Stochastic) Search Techniques	24
2.2.1	Simulated Annealing Algorithm	24
2.2.2	Genetic Algorithm	25
2.3	Illustrative Example	28
2.3.1	Gradient-Based Techniques	29
2.3.2	Genetic Algorithm with Binary String Representation	31
2.3.3	Evolutionary Algorithm with Object Representation	35
2.4	Concluding Remarks	39
3	Advanced Analysis Based Optimization Problem	42
3.1	Methods of Advanced Analysis	42
3.1.1	Inelastic Analysis Algorithm	44

3.1.2	Imperfections	46
3.1.3	Connections Model	47
3.1.4	Local and Global Stability Considerations	48
	<i>Local Buckling</i>	
	<i>Lateral-Torsional Buckling</i>	
	<i>Axial-Flexural Buckling</i>	
3.2	Formation of the Optimization Problem	50
3.2.1	Objective Function	51
3.2.2	Serviceability Criteria	52
	<i>Attainment of Service Loading</i>	
	<i>Connection Rotation Limits</i>	
	<i>Deflections Constraints</i>	
	<i>Yielding Constraint</i>	
3.2.3	Strength Criteria	55
	<i>Attainment of Ultimate Loading</i>	
	<i>Connection Rotation Constraints</i>	
	<i>Local and Member Instabilities Constraints</i>	
	<i>Plastic Hinge Rotation Limits</i>	
3.2.4	Designer Preference Constraints	58
3.3	Concluding Remarks	59
4	Evolutionary Design Algorithm	60
4.1	Python Terminology	60
4.2	Algorithm Overview and Assumptions	62
4.3	Algorithm Details	63
4.3.1	Initialization Module	66
4.3.2	Evaluation Module	73
	<i>Analysis Methods</i>	
	<i>Penalty Methods</i>	

4.3.3	Selection Module	82
	<i>Roulette Wheel Selection</i>	
	<i>Tournament Selection</i>	
4.3.4	Reproduction Module	86
	<i>Crossover</i>	
	<i>Mutation</i>	
4.4	Concluding Remarks	91
5	Frame Design Examples	92
5.1	Frame 1 - Three-Story, Two-Bay	92
5.2	Frame 2 - Two-Story, Three-Bay	101
5.3	Frame 3 - Ten-Story, Three-Bay	116
5.4	Concluding Remarks	126
6	Summary, Conclusions, and Future Work	128
6.1	Summary	129
6.2	Conclusions	130
6.3	Recommendations for Future Work	131
	References	133
	Appendix A	141
	Appendix B	146
	Appendix C	180

List of Figures

1.1	Class Definitions for Object-Oriented Programming Example	12
2.1	Bound and Branch Search Trees	23
2.2	Qualitative Example of Encoding/Decoding of Binary String Representation	27
2.3	Population of Individuals with Binary String Representation	32
2.4	Crossover Operations for Binary String Representation	34
2.5	Population of Individuals with Object Representation	36
2.6	Crossover Operations for Object Representation	37
3.1	Fiber Element Model	44
3.2	Non-Dimensional Connection Curves	48
4.1	Example of Required Input and Evolved Output for the Proposed Algorithm	64
4.2	Schematic Flow Chart of the Proposed Algorithm	67
4.3	Object-Oriented Tree Representation of the Design Variables	68
4.4	Initialization Module Class Hierarchy	70
4.5	Design Variable Options	72
4.6	Scaling Functions	75
4.7	Selection Module Class and Method Hierarchy	82
4.8	Roulette Wheel Selection Scheme	84
4.9	Group Selection (Partitioning) Scheme	85
4.10	Reproduction Module Class and Method Hierarchy	87
4.11	Crossover Operations	88

5.1	Topology and Loading for Frame 1	93
5.2	Fitness Trajectories for Frame 1	94
5.3	Evolved Configurations for Frame 1 with FR Connections	96
5.4	Evolved Configurations for Frame 1 with PR Connections	97
5.5	Assessment of Evaluation Parameters for Frame 1	98
5.6	Load Deformation Response for Frame 1	100
5.7	Service Penalty Convergence for Frame 1	100
5.8	Ultimate Penalty Convergence for Frame 1	101
5.9	Topology and Loading for Frame 2	102
5.10	Fitness Trajectories for Frame 2	104
5.11	Evolved Configurations for Frame 2 with Grade 36 Steel and FR Connections	105
5.12	Evolved Configurations for Frame 2 with Grade 36 Steel and PR Connections	106
5.13	Evolved Configurations for Frame 2 with Grade 50 Steel and FR Connections	106
5.14	Evolved Configurations for Frame 2 with Grade 50 Steel and PR Connections	107
5.15	Evolved Configurations for Frame 2 with Grade 50 Steel, FR Connections, and 12 ft. 1 st Story Height	107
5.16:	Assessment of Evaluation Parameters for Frame 2	108
5.17	Load Deformation Response for Frame 2 with FR Connections	110
5.18	Load Deformation Response for Frame 2 with PR Connections	111
5.19	Load Deformation Response for Frame 2 with Grade 36 Steel	112
5.20	Service Penalty Convergence for Frame 2 with Grade 36 Steel	113
5.21	Ultimate Penalty Convergence for Frame 2 with Grade 36 Steel	113

5.22	Service Penalty Convergence for Frame 2 with Grade 50 Steel	114
5.23	Ultimate Penalty Convergence for Frame 2 with Grade 50 Steel	114
5.24	Load Deformation Response for Frame 2 with Grade 50 Steel and FR Connections	115
5.25	Service Penalty Convergence for Frame 2 with Grade 50 Steel	116
5.26	Ultimate Penalty Convergence for Frame 2 with Grade 50 Steel	116
5.27	Topology and Loading for Frame 3	117
5.28	Fitness Trajectories for Frame 3	119
5.29	Evolved Configurations for Frame 3 with FR Connections	120
5.30	Evolved Configurations for Frame 3 with PR Connections	121
5.31	Assessment of Evaluation Parameters for Frame 3	122
5.32	Load Deformation Response for Frame 3	124
5.33	Service Penalty Convergence for Frame 3	125
5.34	Ultimate Penalty Convergence for Frame 3	126

List of Tables

2.1	Effect of Population Size on the Success of the OO-EA	38
2.2	Effect of Mutation Rate on the Success of the OO-EA	38
2.3	Effect of Non-homologous Crossover Rate on the Success of the OO-EA	39
3.1	Multi-Linear Connection Modeling Data	48
3.2	Connection Weight Modification Factors	52
5.1	Design Parameters for Frame 1	93
5.2	Weight Comparison for Frame 1	99
5.3	Design Parameters for Frame 2	102
5.4	Weight Comparison for Frame 2	109
5.5	Design Parameters for Frame 3	118
5.6	Weight Comparison for Frame 3	124

Chapter 1 – Introduction

The past several decades have seen considerable advancements in the analysis and optimum design of steel frameworks. These advancements have escalated over the past decade with the increased processing speeds of the desktop computer. As a result, researchers are exploring new methods of incorporating highly complex and computational methods of analysis with techniques of optimization, ultimately providing designers with the means to produce more efficient and economical designs.

1.1 Objective and Scope

The objective of this research is to develop an optimization algorithm for the design of unbraced partially restrained (PR) and fully-restrained (FR) steel frames. The design is limited to frames with known topology, loading, and material properties. The algorithm will introduce a new object-oriented representation of the structural components (*e.g.* members, connections). The member sizes will be selected from a database of available AISC wide flanged sections and the beam connections will be selected from a pre-defined non-dimensional list of connections ranging from rigid to flexible. The algorithm will evaluate combinations of member sizes and connections using an advanced (inelastic) analysis program. Upon completion, the algorithm will provide an optimized steel frame design for the prescribed loading condition, performance criteria, and frame topology.

1.2 Background and Literature Review

The focus of this thesis is to incorporate techniques of optimization, advanced analysis, and object-oriented programming (OOP) in the design of steel frames. There has been extensive research and application of these topics in structural engineering. This section reviews optimization techniques and methods of advanced analysis specific to the design of steel frames. Also, a brief introduction to the OOP paradigm will discuss basic object-oriented terminology and concepts used in this thesis.

1.2.1 Structural Optimization

The design of steel frames requires a systematic sequence of (sometimes) laborious trial-and-error steps.

1. An initial trial design is assumed based on the given parameters of the frame (*e.g.* frame topology, loading conditions, and material properties).
2. The response of the frame is determined via structural analysis (inelastic or elastic).
3. The frame response is evaluated with respect to governing design specifications.
4. A new design is selected to eliminate any violations of the specifications and/or improve the economy of the frame.

The trial-and-error process is continued until an acceptable design is attained. The efficiency of the trial and error process is greatly dependent on the experience and ability of the designer to select a “good” initial design and perturb the design in a “better” direction towards an optimal design (Grierson 1997). As a result, techniques of optimization have developed to automate the trial and error procedure.

Optimization techniques can be classified as gradient-based (requiring continuous variable representation) and direct search techniques (requiring discrete variable representation). The background and use of these techniques in the literature will be discussed below with the details of each technique provided in Chapter 2.

1.2.1.1 Gradient-Based Techniques

Gradient-based techniques rely on gradients to guide the algorithm through the solution space. Therefore, the algorithms require continuous functions representing the objective, constraint(s), and design variables defining the optimization problem.

Two major obstacles are encountered when gradient-based techniques are applied to the optimal design of steel frames. First, the design of steel frames is a discrete variable optimization problem by nature, since the variables can only assume standardized shapes. Consequently, researchers using gradient-based techniques often use numerical methods to develop approximate continuous relationships between member properties (*e.g.* moment of inertia) with respect to a single continuous design variable (*e.g.* area) (Thevendran et al. 1992, Al-Salloum 1995; Erbatur and Al-Hussainy 1992). Also, the objective (*e.g.* weight or cost) and constraints (*e.g.* deflection limits) are typically nonlinear and are often classified as nonlinear programming (NLP) problems. As result, the original nonlinear problems are transformed to more manageable sub-problems and approximate solutions are obtained iteratively. Arora (1997) and Arora and Huang (1996) presented several methods to solve the sub-problems. Additional research is cited below.

Gradient-based techniques have been used in the minimum weight design of steel frames using continuous variables and subsequent selection of discrete sections based on the continuous solution. Al-Saadoun and Arora (1989) optimized steel frames for minimum weight design under multiple loading conditions using a multiple design variable approach. Available discrete sections were selected based on a solution obtained using sequential quadratic programming (SQP) algorithm. Chan et al. (1995) used a similar procedure for the optimal design of tall steel building frameworks using a single design variable formation.

Linearization approaches have been used to solve NLP problems. The plastic design of steel frames has been formulated as a linear programming (LP) problem through the use of linearized relationships between weight and the plastic moment of the members and solved using the simplex method (Dixon and O'Brien 1994; Pezeshk 1997). Erbatur and Al-Hussainy (1992) reduced the minimum weight design of steel frames within the elastic range of material behavior to a LP problem.

Liebman (1981) and Thevendran et al. (1992) have converted the constrained NLP problem into an equivalent unconstrained minimization sub-problem using penalty functions. Liebman (1981) used an integer gradient direction method and Thevendran et al. (1992) applied the sequential unconstrained minimization technique (SUMT) to the minimum weight design of multi-story and multi-bay steel frames.

Researchers have studied ways to incorporate connection considerations into optimization algorithms. Simoes (1996), Xu and Grierson (1993), and Xu et al. (1995) investigated the optimal design of partially-restrained steel frames using a dual method optimization algorithm where the member sizes were represented as discrete variables

and the connection stiffness were treated as continuous variables. Al-Salloum and Almusallam (1995) studied the optimal design of partially-restrained steel frames by fixing the connection stiffness and solving for the member sizes using a predictor-corrector optimization algorithm.

Methods have been developed to reduce the extensive computational effort for the analyses required by iterative techniques. The optimality criteria (OC) method reduces the computational effort using an iterative procedure with criteria based on the behavior of the structure under consideration. The optimality criteria (OC) method has been used in the design of steel frames considering geometrically nonlinear elastic-plastic behavior (Hayalioglu and Saka 1992; Saka and Hayalioglu 1991). The OC approach has been used for the optimal design of steel frames using the displacement and combined stress limitations, which include in plane and lateral buckling of members (Saka 1991). Other researchers have developed systematic design procedures that coordinate methods of structural analysis, sensitivity analysis, and optimization techniques into an iterative design process (Chan 1997; Chan et al. 1995; Grierson 1997; Grierson and Chan 1993; Hall et al. 1989). Al-Salloum (1995) used a similar procedure based on the fully-stressed design concept applied to indeterminate elastic frames.

Another method used to reduce the computational effort is the bound and branch method, which is a variation of a discrete exhaustive search technique. The bound and branch method uses continuous solutions to sub-problems to eliminate portions of the solution space represented by search tree. Hager and Balling (1988) selected optimum members of a steel frame by linearizing the objective constraint functions about the continuous optimum and employing a modified bound and branch method. May and

Balling (1991) applied a bound and branch method to the discrete optimization of three-dimensional steel frames.

1.2.1.2 Direct (Stochastic) Search Techniques

Stochastic search algorithms employ random number generation to directly search the solution space. As a result, they do not require evaluation of gradients of objective and constraint functions. Simulated annealing (SA) and the genetic algorithm (GA) are two popular stochastic algorithms that simulate natural phenomena related to the annealing of metal and natural genetic evolution.

The literature on the application of SA specific to steel frame optimizations is relatively sparse compared to the GA. Balling (1991) used the SA method coupled with a linked discrete variable whose value (*e.g.* standard section designation) specifies the values of a group of parameters related to it (*e.g.* cross-sectional area, moment of inertia, etc.). May and Balling (1992) applied a filtered SA strategy that probabilistically filtered out poor candidate designs based on known gradient information.

The GA was introduced to the civil engineering community by Goldberg and Samtani (1986) and was immediately embraced by structural optimization researchers. Over the past decade, the GA has been readily applied to a broad range of structural applications from a cable-stayed bridge (Jenkins 1992) to a 160-bar transmission tower (Rajeev and Krishnamoorthy 1992). Leite and Topping (1998) provides an extensive discussion of a wide range of applications of the GA including performance comparisons with other optimization methods.

Much of the GA research applied to structures has focused on trusses allowing for comparison studies of various types of GAs and expansion of GAs to solve for the optimal topology. Jingui et al. (1996), Rajeev and Krishnamoorthy (1992), Rajeev and Krishnamoorthy (1997) developed GAs to solve for the minimum weight design of trusses assessing their performance using the 10-bar truss initially studied by Goldberg and Samtani (1986). Yang and Soh (1997) assessed GA selection mechanisms by solving planar and space truss optimization problems and concluded that competitive selection schemes (*i.e.* tournament) out performed fitness-proportional selection schemes (*i.e.* roulette wheel). GA's have been developed to optimize trusses by simultaneously considering size, shape, and topology (Rajan 1995; Shrestha and Ghaboussi 1998).

There have been a limited number of applications of the GA to steel frames. (Camp et al. 1996; Camp et al. 1998; Hayalioglu 2000; Kameshki and Saka 1999; Pezeshk et al. 1997; Pezeshk et al. 2000). Camp et al. (1996) developed a design procedure (FEAPGEN) that incorporates a GA, a finite element analysis program, and a complete database of AISC sections for the design of steel frames according to AISC-ASD. Comparison studies of the design of a one-bay, eight-story frame and a three-bay, three-story frame found FEAPGEN resulted in lighter frames compared to a traditional optimality criteria (OC) algorithm.

Pezeshk et al. (1997) incorporated a group selection scheme, an adaptive crossover scheme, and finite element analysis into a GA to design frames according to AISC-LRFD and AISC-ASD. The steel frame design according to AISC-LRFD resulted in a slightly lighter frame than a design according to AISC-ASD. Furthermore, the AISC-LRFD design and AISC-ASD design were significantly (4 to 7 %) lighter than a

design using a traditional OC algorithm. However, the GA required significantly more computational time than the OC algorithm.

Kameshki and Saka (1999) studied the effects of semi-rigid connections in the minimum weight design of frames using a GA that incorporates both the geometric nonlinear behavior of the frames and the nonlinear connection response into the method of analysis.

1.2.2 Advanced Analysis

Conventional methods of limit state design incorporate two primary means to simplify the steel frame model: (a) the implicit accounting of nonlinear material and geometric behavior and (b) the idealization of beam-to-column connections as pinned or rigid.

Since limit state design requires consideration of structural behavior as the structure approaches its limit of resistance, the analysis needs to account for the nonlinear response that may be experienced by most steel structures. Specifications approximate the inelastic behavior by modifying an elastic analysis with empirical allowances for non-linearity. However, computationally rigorous methods of inelastic analysis based design, termed advanced analysis, previously deemed computationally infeasible with traditional computers, are becoming accepted methods of design (ABCB 1998, CEN 1993). The AISC (1993) specification equations used to assess member strength are based upon advanced analytical formulations calibrated with experimental data. With the advent of increased computational power comes the ability to take these same advanced formulations and include them in inelastic structural analysis packages. As a result,

computer models now have the capability of performing a structural analysis that captures the pertinent structural behavior used in the development of the design specification equations, therefore, eliminating the need for specification equations.

The aim of advanced analysis based design is to perform an inelastic analysis for frame strength assessment that includes the spirit and assumptions of the design equations contained in specifications. Ziemian (1992) studied the limitations of various methods of advanced analysis in the limit state design of steel structures and compared results from advanced analysis methods with conventional methods of elastic analysis and code equations. The majority of advanced analysis research has focused on the plastic hinge approach (White 1993, White 1992) and refined plastic hinge approach (Liew and Chen 1993a, Liew and Chen 1993b, Liew et al. 1993a, Liew et al. 1993b, Kim and Chen 1996b, Kim and Chen 1996a). Although the assumptions established by these approaches greatly simplify the computational effort, the accuracy of the structural response suffers significantly relative to the more computationally rigorous distributed plasticity or plastic zone approach (Clarke et al. 1992, Foley 1996).

Research has shown that the actual stiffness or restraint of connections lies between the two extremes of pinned and rigid, resulting in the development of connection stiffness models (Bjorhovde et al. 1990). Barakat and Chen (1990) incorporated connection stiffness considerations into the established AISC-LRFD design approach using two proposed connection models. The first model uses a modified initial stiffness representation and the second model is determined by the beam-line method. Xu et al. (1995) used empirical data to represent the stiffness of partially-restrained (PR) connections. However, the resulting moment-rotation curves offered little practical value

to the designer, since connection behavior (*i.e.* strength and stiffness) is highly dependent on the connecting members. Thus, there was a need for non-dimensional connection moment-rotation relationships (Bjorhovde et al. 1990).

The continued development of efficient and accurate methods of advanced analysis and practical connection models has shown promise for true performance-based specifications for steel frame design in lieu of the conventional specification equations. Furthermore, the current computational technology has proven capable to support computationally intensive models of analysis and design (Foley 1996).

1.2.3 Object-Oriented Programming

A primary objective of this research is the application of object-oriented programming (OOP) to the design of steel frames. This section provides a brief review of OOP concepts and terminology used in this thesis and some previous applications of OOP to structural design.

According to Ambler (1995), the object-oriented (OO) paradigm is a strategy based on constructing systems (*i.e.* algorithms) as a collection of reusable components or *objects*. The OO paradigm used by contemporary programming languages such as C++ and Python differs from the structured paradigm used by conventional programming languages such as FORTRAN or C. The structured paradigm separates the system into two parts: data and functionality. In other words, a structured paradigm distinctly treats the layout of a data model (*i.e.* database) and design of a process model to access the database (*i.e.* functionality) as separate parts. In contrast, the OO paradigm defines systems as a collection of interacting *objects* possessing both functionality and data. An

object can represent a person, place, thing, concept, or event applicable to the system. Instead of writing a program to access a database, OOP implements an *object space* where both the program and the data coexist.

The *object space* can be viewed as a place where *objects* reside in an organized hierarchy of *modules* and *classes*. Furthermore, each *object* possesses *attributes* and *methods*. Consider a simple example of a computerized banking system involving customers making transactions. The structured paradigm creates a database and an accompanying program to access and manage the database, whereas the OO paradigm identifies the key components and represents them as *objects* with *attributes*. The *object space* is divided into *modules* representing branches or banking locations. Furthermore, each *module* contains *classes* or a collection of similar *objects*. These *objects* can correspond to anything from a teller to a deposit transaction and are assigned characteristics (*attributes*) and behavior (*methods*) according to their *class*. This simple example considers a customer class and a transaction class as shown in Figure 1.1. Suppose a transaction is carried out at one of the banking locations. In OO terminology an *instance* of the transaction class is established by *instantiating* (or creating) a transaction *object*. This new transaction *object* is assigned default behavior (*i.e.* the ability to credit a customer account in the event of a deposit) and characteristics (*i.e.* an identification or transaction number for future reference). Another way to view this is to associate the *class* as a table or database and the *object* as a record occurrence; however, unlike a database, which defines only data, a *class* defines both data (*attributes*) and code (*methods*).

Abstraction, encapsulation, and polymorphism are concepts used to develop the OO paradigm. Each of these concepts can be explained with respect to the OO banking system. *Abstraction* determines what a *class* needs to know about an *object*. For example consider the *abstraction* of a customer. The bank requires certain record keeping information including the customer's name, social security number, address, and phone number. However, each customer has additional characteristics not required by the bank including the customer's weight, hair color, and hobbies. In other words, *abstraction* defines what a *class* needs to know and do with respect to the *objects* it *instantiates*.

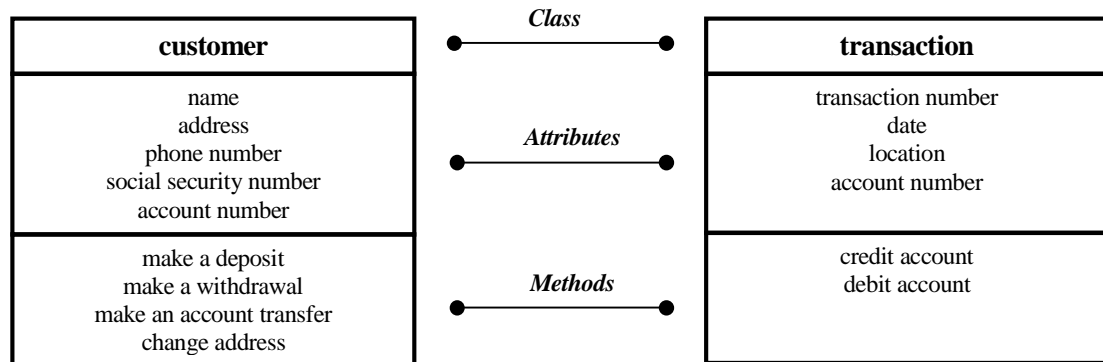


Figure 1.1: Class Definitions for Object-Oriented Programming Example

Encapsulation defines how functionality is compartmentalized within a system. In OO terminology, the banking system is modularized into a customer *class* and a transaction *class*, in which the customer behavior is *encapsulated* into the customer *class*. Furthermore, the functionality of making a deposit is *encapsulated* into the “make a deposit” *method* contained in the customer *class*. *Encapsulation* is often referred to as a

black box, since the *objects* are identified without specifically defining the implementation. However, this is the beauty of OOP, since the utilization of an *object* is independent of its implementation. For example, suppose the bank decides to change how it processes transactions, it should not affect the service to the customer (as long as the bank's records are consistent with the customer's records).

The concept of *polymorphism* allows *instances* of various *classes* to be treated the same way within the system. In some cases, a system may contain two *classes* with the same *method*. However, two different types of functionality are *encapsulated* depending on which *class* the corresponding *method* resides. For example, suppose another *class* (*i.e.* investment *class*) is added to the banking system allowing customers to open an investment account. Furthermore, the investment *class* contains a “credit account” method. In OO terminology, the functionality of crediting the new investment account is *encapsulated* into the “credit account” *method* contained in the investment *class*. However, recall that the “credit account” *method* is also *encapsulated* in the transaction *class* as seen in Figure 1.1. Through *polymorphism* the OO banking system is able to distinguish between the “credit account” *methods* and ensures that the correct accounts are credited.

The previous overview of OO terminology and concepts is limited to those utilized within this thesis and is by no means complete. The OO paradigm is an extremely useful tool with several additional applications and capabilities. The civil engineering profession has recognized the power of the OOP paradigm and the following are examples of research efforts that incorporate OOP methods into structural design.

At the structural member level, OOP has been applied to both steel and wood beam-column design (Garrett and Hakim 1992; Voss 1992). Biedermann (1996) outlines an object-oriented prototype to manage the complexity involved in programming the design process for building systems. An object-oriented “front-end” was written to drive the design of structural steel building frameworks using a structural optimization design & analysis program (SODA 1999). Modifications to the “front-end” were then discussed and applied to the optimal design of timber systems using WoodWorks (2000). This research has shown OOP to be very efficient as a design *wrapper* program for previously written procedural code, which has very important ramifications for the present research.

Object-oriented approaches have also shown promise in implementing highly complex models (product and design) of engineering design (Sause et al. 1992). The *abstraction* and *encapsulation* of data with procedures offered by OOP makes it possible to unify these two models of the engineering design process. Rigopoulos and Oppenheim (1992) proposed the use of an OO model to represent an evolving structural system design. OOP offered a means to develop inter-object relationships, such as beam *objects* supporting slab *objects*. For example, a slab *object* would know that at least two beam *objects* must be *instantiated* for its support. As a result, supervisor, connector, and delegater *objects* can manage the evolution of building structures (Rigopoulos and Oppenheim 1992). A recent application of OOP in building analysis and design was presented by Rivard and Fenves (2000). They proposed a hierarchical representation of the entire building project, which provides an excellent view into the future of the computerized engineering of buildings.

1.3 Overview of Contents

To accomplish the objectives of this research, an evolutionary design algorithm is developed to optimize the design of FR and PR steel frames using inelastic analysis-based design. In Chapter 2, the methods of optimization are reviewed with an example illustrating the difference between gradient-based and direct search methods. The example also demonstrates the implementation of a unique object representation of the design variables to be used in the present research. The chapter concludes with a justification for using a stochastic direct search optimization algorithm to drive the evolutionary design algorithm.

In Chapter 3, the formation of the advanced analysis based optimization problem is presented for the minimum weight design of unbraced steel frames. The inelastic analysis and connection model will be discussed as well as the necessary out-of-plane considerations for planar analysis. The constraint criteria are developed for serviceability, strength, and constructability considerations.

In Chapter 4, the evolutionary design algorithm is described in detail. The *encapsulation* of each of the primary components of the evolutionary algorithm is explained with reference to OO terminology.

In Chapter 5, the evolutionary design algorithm is implemented using design examples found in the literature. The performance of the proposed algorithm is evaluated with comparisons to previous results.

Finally, Chapter 6 summarizes the research with conclusions and recommendations for future work.

Chapter 2 - Methods of Optimization

Optimal design of steel frames typically involves finding set of discrete design variables (*e.g.* member cross-sectional properties) to minimize an objective function (*e.g.* weight or cost of a steel frame) subject to one or more constraints (*e.g.* performance requirements and/or manufacturing constraints) for a predetermined structural layout and loading (Grierson 1997a). The result is a multi-dimensional constrained minimization problem of the form,

$$\textit{Find} \quad \mathbf{X} = \{x_1, x_2, \dots, x_n\} \quad (2.1)$$

$$\textit{to minimize} \quad f(\mathbf{X}) \quad (2.2)$$

$$\textit{subject to} \quad g_j(\mathbf{X}) = 0 \quad j=1 \text{ to } p \quad (2.3)$$

$$g_j(\mathbf{X}) \leq 0 \quad j=p+1 \text{ to } m \quad (2.4)$$

$$x_i^L \leq x_i \leq x_i^U \quad i=1 \text{ to } n \quad (2.5)$$

where, $f(\mathbf{X})$ is the objective (cost) function, $g_j(\mathbf{X})$ are constraint equations, and x_i^L and x_i^U are the upper and lower bounds of the design variables, \mathbf{X} . The objective is to find a point, \mathbf{X}^* corresponding to the lowest possible value of the objective function without violating the constraints (Rajan 2001).

Small-scale minimization problems can be solved using exhaustive search, graphical techniques, or trial-and-error processes. However, an exhaustive search is too computationally expensive for practical problems; graphical techniques are only practically applicable for problems with at the most two design variables; and the trial-and-error approach is cumbersome with its reliability heavily dependent on the designer.

Typically, optimal designs of steel frames require the use of “constraint controlled” techniques (Rajan 2001).

Researchers have explored several “constraint controlled” techniques of solving constrained minimization problems with discrete design variables (Arora and Huang 1996). The techniques can be divided into gradient-based techniques and direct search techniques.

This chapter will review methods used in the optimal design of steel frames. In addition, an example is presented illustrating the implementation of gradient-based and direct search methods on a simple algebraic function. Furthermore, the example will be used to demonstrate the application of a stochastic direct search method (evolutionary algorithm) developed as the focus of this thesis.

2.1 Gradient-Based Techniques

Gradient-based techniques rely on gradients of the objective function and constraint equation(s) to guide the algorithm to the optimum solution. Consequently, the design variables need to have continuous values and the objective and constraint functions need to be differentiable with respect to the design variables. However, it is difficult to represent the design variables used in steel frame design as continuous, since the cross-sectional shapes of steel sections are manufactured to discrete standard sizes. Furthermore, the connections found in partially-restrained frames are also very difficult (if not impossible) to represent in a continuous manner.

One way to consider a discrete design variable problem with a gradient-based technique is to solve the minimization problem with continuous design variables and

select a set of discrete design variables based on the continuous solution (Al-Saadoun and Arora 1989; Simoes 1996). However, the resulting frame containing the selected set of discrete design variables may not satisfy the imposed constraints of the problem originally satisfied by the continuous solution (Camp et al. 1998).

Another obstacle encountered by gradient-based techniques is the need to continuously represent secondary properties (*e.g.* moment of inertia, radius of gyration, etc.) often required by constraint equations. Many gradient-based techniques use a single primary continuous design variable and relate other secondary design variables to this continuous design variable through functions. For example, the primary design variable could be the cross-section area, A , and a secondary design variable would then be the second moment of area, I where,

$$I = f(A)$$

There could be multiple secondary design variables, such as radius of gyration, section modulus, etc. Additional functions relating these properties are required, and examples of this process are available in the literature (Al-Salloum 1995; Al-Salloum and Almusallam 1995; Chan 1997; Erbatur and Al-Hussainy 1992; Grierson 1997a; Thevendran et al. 1992). It can be difficult to analytically express all cross-sectional properties as functions of a single cross-sectional property for any given member. For example, many of the rolled steel sections manufactured in the U.S. have similar cross-sectional areas, however, the other properties are significantly different.

2.1.1 Nonlinear Programming Problem

In the optimal design of steel frames, the objective and constraint equations expressed in (2.2) through (2.4) are frequently nonlinear resulting in a nonlinear programming (NLP) problem. The constrained NLP problem can be transformed into a series of unconstrained problems (Arora 1997). A common means to accomplish this transformation is to define a Lagrangian function of the form,

$$L(\mathbf{X}, u) = f(\mathbf{X}) + \sum_{j=1}^m u_j \cdot g_j(\mathbf{X}) \quad (2.6)$$

where, the constraints, $g_j(\mathbf{X})$, can represent both equality, $g(\mathbf{X}) = 0$, and inequality, $g(\mathbf{X}) \leq 0$ constraints and u_j are referred to as Lagrange multipliers. The Lagrangian, $L(\mathbf{X}, u)$, is minimized rather than the original function, $f(\mathbf{X})$.

In order to guarantee a solution, the number of equality constraints must not exceed the number of variables. In addition, the design variables need to have continuous values and the objective and constraint functions need to be differentiable with respect to the continuous design variables. A stationary point or relative minimum, \mathbf{X}^* exists if,

$$\frac{\partial L(\mathbf{X})}{\partial \mathbf{X}} = \frac{\partial f(\mathbf{X})}{\partial \mathbf{X}} + \sum_{j=1}^m u_j \cdot \frac{\partial g_j(\mathbf{X})}{\partial \mathbf{X}_i} = 0 \quad \forall \quad i = 1, 2, \dots, n \quad j = 1, 2, \dots, m \quad (2.7)$$

The above $(n + m)$ equations are known as the Kuhn-Tucker necessary conditions.

Solutions satisfying the Kuhn-Tucker conditions are candidate local minimums unless the problem can be shown to be convex in which case they are also the sufficient conditions for the candidates to be a global minimum solution. However, proving the problem to be convex is an extensive procedure. Another way to determine if the design variable vector is a global minimum is to distinguish the local minimum using the Hessian, which is a

matrix comprised of second order partial derivatives of the Lagrangian function (Arora 1997). Typically, the solutions to steel frame design satisfying the Kuhn-Tucker conditions represent local minimums.

An illustrative example provided in a subsequent section will demonstrate that simple NLP problems can be solved “by hand”. However, the multiple constraints and nonlinear nature associated with most optimal steel design problems requires the minimization problem to be solved iteratively using numerical methods. These numerical methods convert the original nonlinear problem to a more manageable sub-problem. However, each iteration typically requires a structural analysis to be performed resulting in an immense computational effort for problems with a large number of design variables. Some popular solution techniques include: sequential linear programming (SLP), sequential quadratic programming (SQP), the feasible directions method, the generalized reduce gradient (GRG), and the sequential unconstrained minimization technique (SUMT) (Rajan 2001). Arora (1997) and Arora and Huang (1996) provide a discussions and references to several of these techniques.

2.1.2 Optimality Criteria

For problems involving a large number of design variables, the iterative structural analysis performed can become computationally extensive. To ease the computational effort, the optimality criteria (OC) transformation method utilizes known characteristics of the problem (Grierson 1997b, Grierson and Chan 1993, Chan et al. 1995, Hayalioglu and Saka 1992, Saka 1991; Saka and Hayalioglu 1991). The OC method formulates a set

of necessary conditions based on the minimization of a Lagrangian function similar to equation (2.6),

$$L(A_i, u_r) = \sum_{i=1}^n w_i \cdot A_i + \sum_{j=1}^m u_j \cdot \left[\sum_{i=1}^n \frac{c_{ir}}{A_i} - \bar{g}_j \right] \quad (2.8)$$

where, u_j are the Lagrange multipliers, w_i are the weight coefficients (*i.e.* member length multiplied by density), A_i are the design variables representing cross-sectional area (*i.e.* the primary design variable), \bar{g}_j are the constraint bounds, and c_{ir} are the constraint gradients developed using a sensitivity analysis of each constraint (*e.g.* displacement, flexural stress, etc.). The gradients are dependent on the class of structure (*e.g.* truss, frame, etc.) and can be approximated using the method of virtual work, pseudo-load method, or a finite difference technique (Grierson 1997a).

The optimality conditions are expressed by differentiating the Lagrangian function with respect to the design variables. These conditions simplify to become,

$$\sum_{j=1}^m u_j \cdot \left[\frac{c_{ir}}{w_i \cdot A_i^2} \right] = 1 \quad (2.9)$$

The above equation represents the necessary conditions for an optimal design. The terms in the optimality conditions have direct meaning within the context of optimal steel design. The Lagrange multipliers measure the sensitivity of the optimal design to each constraint. The expression in the brackets represents the virtual strain energy densities for each member. Therefore, the optimality conditions require the weighted sum of the virtual strain energy densities of each member to equal one. In other words, the optimal design variable vector for a structure subjected to multiple constraints is one where each member of the structure contributes an equal amount of virtual strain energy density

when satisfying a specific constraint. It should be noted at this point, that there are as many optimality criteria as there are constraints in a given problem.

The optimality conditions are solved iteratively for the Lagrange multipliers and the design variables using a recursive algorithm (Chan 1997; Grierson 1997b). It should be noted that for determinate structures, the recursive algorithm results in the optimal solution without the need for reanalysis, since member forces can be calculated independently of the member sizes. However, indeterminate structures require multiple analyses since the solution from the recursive algorithm must be reanalyzed due to the variation in force distribution within the revised member sizes.

2.1.3 Bound and Branch Method

The branch and bound method is considered a combinatorial search method, since it combines a discrete exhaustive search with a continuous variable, gradient-based technique (*e.g.* SLP, SQP, etc.) to reduce the solution space. It is best described using a modified version of the qualitative example developed by (Balling 1997).

Consider a minimization problem involving three design variables, $[x_1, x_2, x_3]$, where each design variable represents a value in the finite set of positive integers, $\{A, B, C\}$. The goal is to minimize the objective function subjected to constraints.

Branch and bound search trees can be constructed as illustrated in Figure 2.1. Each level in the tree represents a design variable and each rectangle represents a node. Within each node are three values that represent the node number, the discrete value of the variable, and the objective function bound (*i.e.* the evaluated value of the objective function),

respectively. The nodes are numbered in the order they are generated. A solution that is not feasible (*i.e.* violates a constraint) is denoted with an asterisk.

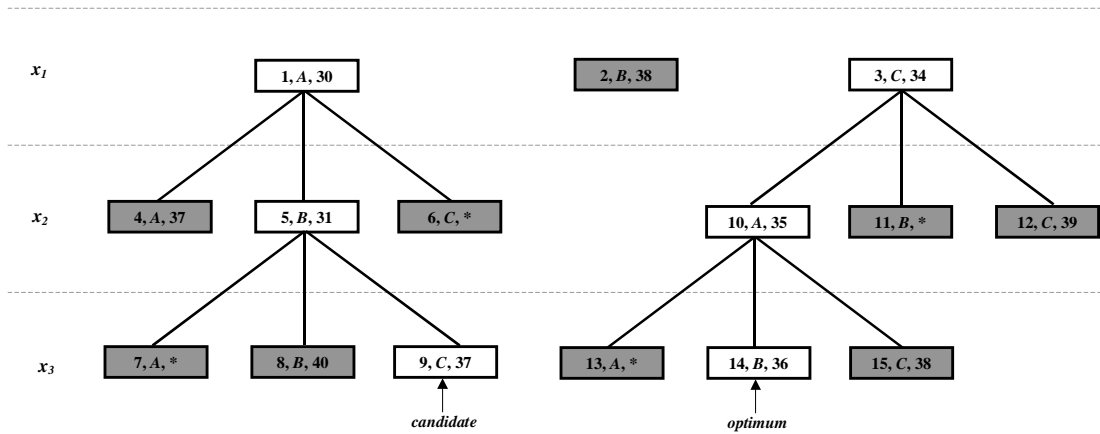


Figure 2.1: Bound and Branch Search Tree

The first step is to generate *node 1* by setting $x_1 = A$ while treating x_2 and x_3 as continuous variables. Using a gradient-based technique, the objective function is evaluated to be 30. This represents a lower bound on the objective function since any combination of x_2 and x_3 together with the fixed value of $x_1 = A$ would result in a larger value of objective function. Similarly, the same procedure can be done for *node 2* and *node 3*, where x_1 is assigned values of B and C and a gradient-based technique results in objective function values of 38 and 34, respectively.

The next step is to expand *node 1* since it resulted in the lowest objective function bound. *Node 4*, *node 5*, and *node 6* are generated by repeating the first step and maintaining $x_1 = A$. The problem is solved via a gradient-based technique for fixed discrete values of x_2 and continuous values of x_3 . The procedure continues until the

branch is completed resulting in a *candidate* solution, $\{A, B, C\}$ (see Figure 2.1). The candidate solution is used to eliminate or “fathom” the nodes with higher objective function bounds depicted with darkened nodes in Figure 2.1.

The process continues by expanding any remaining nodes. After each step down the tree, nodes resulting in higher objective function bounds are “fathomed” and nodes resulting in lower objective function bounds are expanded until all the branches are explored. In the example, *node 14* corresponds to the optimal solution $\{C, A, B\}$ resulting in an objective function value of 36.

2.2 Direct (Stochastic) Search Methods

Stochastic search algorithms are direct search techniques that explore the design space by generating random numbers to guide the algorithm to an optimal design. Stochastic search algorithms utilize discrete design variables and do not require gradients of the objective and constraint functions; however, they are typically more computationally intensive with respect to gradient-based NLP techniques. Simulated annealing and the genetic algorithm are random search techniques that imitate processes found in nature.

2.2.1 Simulated Annealing Algorithm

The simulated annealing algorithm is an iterative process simulating the phenomena of the annealing of metal as it cools from liquid to solid states. The process consists of iteratively evaluating the objective function and constraint equation(s). It is guided through the solution space by means of random number generation, in which variables are “perturbed” incrementally to higher and lower values depending on the random

numbers generated. Typical random search algorithms have a bias to only step in the direction of decreasing the objective function (assuming the optimization problem is a minimization problem). However, to prevent the simulated annealing algorithm from converging to a non-optimal solution, a probability of accepting a higher value of objective function is incorporated. This is accomplished using a probability function,

$$P = \exp\left(\frac{f(\mathbf{X}^*) - f(\mathbf{X})}{T}\right) \quad (2.10)$$

where, $f(\mathbf{X}^*)$ and $f(\mathbf{X})$ are the values of the objective function for the previous solution, \mathbf{X}^* and the current solution, \mathbf{X} . The temperature, T , is reduced each iteration from an initial temperature, T_i : $T = \alpha T_i$ with $0 \leq \alpha \leq 1$. This allows the algorithm to accept higher values of the optimization function in the initial iterations, but is less likely to accept higher values of the optimization function in later iterations. This is similar to the behavior of metal when it transforms from a liquid to a solid. The simulated annealing at a “slow” rate (*i.e.* gradually reduced temperature) gives the solution time to organize itself into the form of the global minimum, much like metal when cooled seeks a minimum energy configuration and forms an organized packing of its atoms. When rapid cooling occurs, the form the metal takes is amorphous (highly unorganized crystalline structure) and it will solidify into a glass state equivalent to a local minimum.

2.2.2 Genetic Algorithms

The genetic algorithm (GA) is a direct search procedure based on Darwin’s survival of the fittest theory (Holland 1975). Characteristics that distinguish the GA from gradient-based techniques include (Goldberg 1989):

- a.) The GA does not operate directly with the design variables, but with a coding of the design variables (typically binary string representations).
- b.) The GA only requires evaluation of the objective function and does not require gradients of the object or constraint equations.
- c.) The GA evaluates the objective function for several randomly selected design points rather than sequentially stepping from one point to the next based on gradient information (*i.e.* hill climbing).

The GA simulates the rules of natural genetic evolution by systematically applying selection, evaluation, crossover, and mutation operations. A population of individuals is generated and the genetic make-up of each individual is constructed by encoding its design variables (or genes) into a single binary string called a chromosome. Figure 2.2 illustrates the encoding/decoding of binary strings representing the design variables (W-shapes) of a portal frame. The GA establishes a fitness function that penalizes individuals that violate the constraints of the problem (*i.e.* column deflection limits). The fitness of each individual is evaluated and individuals are selected based on fitness to form a mating pool. The individuals in the mating pool are altered using crossover (*i.e.* exchanging of portions of binary strings) and mutation (*i.e.* random changing of binary bits) operations resulting in a new population of individuals (*i.e.* offspring). The new population is carried over to the next generation. The fitness of each individual in the new population is determined and the process continues until an established convergence criteria is achieved or a specified number of generations have elapsed and the fittest individual wins. It is important to note that the GA searches for the best (fittest) individual within the population. However, this does not guarantee selection

of the best possible (optimal) individual, since it may not be present in the population (Erbatur et al. 2000).

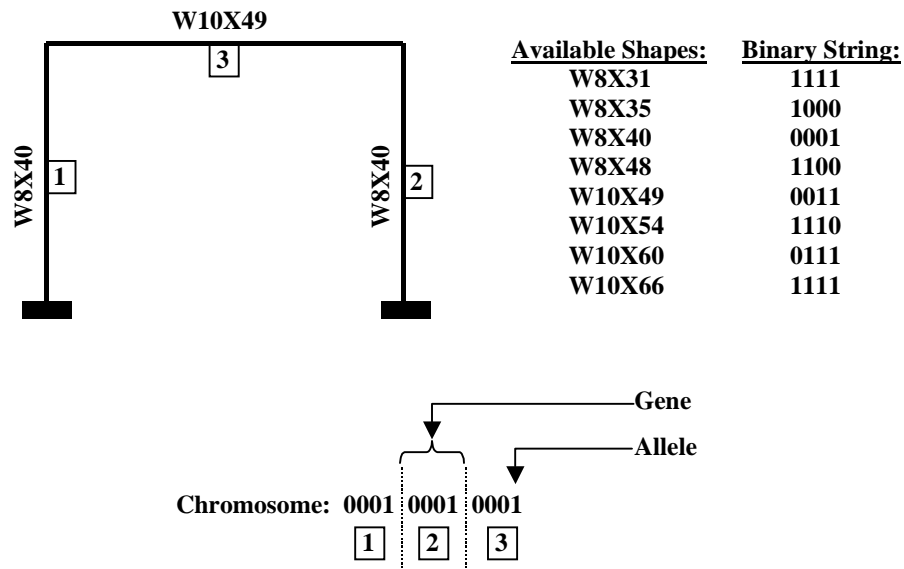


Figure 2.2: Qualitative Example of Encoding/Decoding of Binary String Representation

Spears et al. (1993) classify the GA as the latest variation in the broader class of evolutionary computation models or evolutionary algorithms (EAs). Although similar in basic conceptual framework, the GA differs from the EAs in the representation of the design variables and the implementation of the basic operations (*e.g.* selection, crossover, and mutation). In general, the GA can be applied to various types of problems, since it uses a more versatile design variable representation, namely binary strings, whereas EAs typically use a real-valued vector representation of the design variables tailored to a specific problem. Also, the GA uses crossover as the primary search operator and mutation as a secondary operation, while EAs rely heavily on adaptive mutation, allowing the rate or extent of mutation to vary generationally. In fact some EAs do not

use crossover at all. Furthermore, the GA typically only carries the offspring from the mating pool to the next generation, where EAs offer the opportunity for a combination of offspring and parents to enter the next generation.

Researchers are continually developing new variations of EAs. Voss and Foley (1999) have proposed an EA with a heuristic tree representation of the design variables. This approach uses *a priori* knowledge of the problem to represent design variables as ordered blocks of bits, which laid the groundwork for the research developed in this thesis. The present research effort takes the concept of ordered blocks and establishes the design variables as objects using object-oriented programming (OOP). Just as with previously developed representations, the impact of the object representation on the application and performance of the genetic/evolutionary operations needs to be analyzed. This will be addressed within the illustrative example presented in the next section.

2.3 Illustrative Example

Consider the following constrained minimization problem involving a simple algebraic function subject to three constraints,

$$\textit{Find} \quad \mathbf{X} = [x_1 \ x_2] \quad (2.11)$$

$$\textit{to minimize} \quad y(\mathbf{X}) = \frac{3}{4} \cdot x_1^2 + x_1 \cdot x_2 + \frac{5}{4} \cdot x_2^2 \quad (2.12)$$

$$\textit{subject to} \quad x_1 > 0 \quad (2.13)$$

$$x_2 > 0 \quad (2.14)$$

$$x_1 + x_2 \geq 8 \quad (2.15)$$

The solution to this minimization problem will be demonstrated using a gradient-based NLP technique, a genetic algorithm with binary representation of the design variables, and an object-oriented evolutionary algorithm. The objective of this example is to:

- a.) Demonstrate the use of gradient-based methods and direct search methods.
- b.) Present the implementation of binary versus object representation of the design variables.
- c.) Study the effect of various parameters (*e.g.* population size, crossover rate, mutation rate) on the performance of an object-oriented evolutionary algorithm in solving a simple problem.

2.3.1 Gradient-Based Technique

Since the problem is simple, it can be solved directly by explicitly formulating and satisfying the Kuhn-Tucker (K-T) Conditions via case-by-case evaluation. First, the objective function is written in the form of the Lagrangian,

$$L(x_1, x_2, u_1) = \frac{3}{4} \cdot x_1^2 + x_1 \cdot x_2 + \frac{5}{4} \cdot x_2^2 + u_1 \cdot (-x_1 - x_2 + 8) \quad (2.16)$$

Note that the third constraint is transformed into the form,

$$g_1 = -x_1 - x_2 + 8 \leq 0 \quad (2.17)$$

Also, the Lagrangian does not include the first two constraints (2.13) and (2.14) requiring the variables to be zero or positive. These constraints will be used to select the optimal solution. The optimal solution must satisfy the K-T necessary conditions,

$$\frac{\partial L(x_1, x_2, u_1)}{\partial x_1} = 0 = \frac{3}{2} \cdot x_1 + x_2 - u_1 \quad (2.18)$$

$$\frac{\partial L(x_1, x_2, u_1)}{\partial x_2} = 0 = x_1 + \frac{5}{2} \cdot x_2 - u_1 \quad (2.19)$$

$$u_1 \cdot (-x_1 - x_2 + 8) = 0 \quad (2.20)$$

$$-x_1 - x_2 + 8 \leq 0 \quad (2.21)$$

$$u_1 \geq 0 \quad (2.22)$$

Equation (2.20) represents a switching condition that requires either u_1 or g_1 (the expression inside the parentheses) to be zero. The solutions for both conditions will be solved in Cases A and B below.

Case A: ($\mu_1 = 0$)

$$\left. \begin{array}{l} 0 = \frac{3}{2} \cdot x_1 + x_2 - \mu_1 \\ 0 = x_1 + \frac{5}{2} \cdot x_2 - \mu_1 \\ 0 = \mu_1 \end{array} \right\} \Rightarrow x_1 = 0, \quad x_2 = 0$$

The solution $\mathbf{X} = [0 \ 0]$ is invalid, since it violates the constraint equation (2.21).

When the Lagrange multiplier is less than or equal to zero, it is mathematically implied that the constraint is inactive (Rajan 2001). Therefore, Case A in reality is a unconstrained optimization problem with two equations in two unknowns. Furthermore, since the solution to Case A violates the constraint, a second case must be considered where the constraint is active.

Case B: ($g_1 = 0$)

$$\left. \begin{array}{l} 0 = \frac{3}{2} \cdot x_1 + x_2 - \mu_1 \\ 0 = x_1 + \frac{5}{2} \cdot x_2 - \mu_1 \\ 0 = -x_1 - x_2 + 8 \end{array} \right\} \Rightarrow x_1 = 6 \quad , \quad x_2 = 2 \quad , \quad \mu_1 = 11$$

The first two equations in the set of three above involve gradients of the constraints with respect to each of the design variables. Therefore, the Lagrange multiplier is a scaling factor that moves the solution along the constraint boundary. The solution to the three equations satisfies equations (2.21), (2.22), (2.13), and (2.14).

Therefore, $\mathbf{X}^* = [6 \quad 2]$ is a stationary point (*i.e.* one that satisfies the K-T conditions) resulting in a minimum objective function value of $y(\mathbf{X}^*) = 44$.

2.3.2 Genetic Algorithm with Binary String Representation

Jenkins (1991) solved this simple example using a GA with binary representation of the design variables. This section summarizes the work of Jenkins (1991) in a form suitable for comparison with the object-oriented evolutionary algorithm discussion to follow.

An initial population of individuals or solution sets of the two variables,

$\mathbf{X} = [x_1 \quad x_2]$, are generated randomly. The binary representation assigns each variable a string of bits or alleles (1's and 0's) that are concatenated together to form a gene.

Individuals are represented by concatenating the genes together to form chromosomes.

The number of bits required to represent a gene depends on the range of values available to each variable (upper and lower bounds) and the desired precision of the values. A

binary string of four bits is required in this case because the values of x_1 and x_2 are limited to integers between 0 and 15. The binary representation for a population size of “N” individuals is displayed in Figure 2.3.

	x_1	x_2	Genetic Terminology: allele = 1 or 0 genes = x_1 or x_2 = 1101 or 1100 chromosome = x_1x_2 = 11011100
INDIVIDUAL 1	1100	1010	
INDIVIDUAL 2	0110	1111	
INDIVIDUAL 3	1010	0011	
	⋮		
INDIVIDUAL N	1111	0000	

Figure 2.3: Population of Individuals with Binary String Representation

The GA will seek to maximize fitness. Therefore, the fitness function will need to be expressed such that a lower value corresponds to a larger value for the fitness function,

$$\hat{f}(\mathbf{X}) = C - y(\mathbf{X}) \quad (2.23)$$

where: C is a constant with a value large enough to prevent the fitness from taking on a value of zero. This is accomplished by evaluating the objective function for the largest values of x_1 and x_2 (*i.e.* $C = y(15,15) = 675$). GA's require that the minimization problem be unconstrained. Constraints (2.13) and (2.14) are satisfied through the initialization of the population and the upper and lower bounds of the design variables (*i.e.* a lower bound of one and upper bound of 15). Constraint (2.15) is accounted for using a penalty. If a solution set violates the constraint equation then the objective function is assigned a value of C resulting in a fitness of zero.

To assist in the convergence to the optimal solution, the fitness of each individual is scaled using a linear method developed by (Goldberg 1989),

$$f' = \left(\frac{f_{avg}}{f_{max} - f_{avg}} \right) \cdot f + f_{avg} \cdot \left(1 - \frac{f_{avg}}{f_{max} - f_{avg}} \right) \quad (2.24)$$

where, f' is the scaled fitness, f_{avg} is the average fitness of the generation, f_{max} is the maximum fitness of the generation, and f is the unscaled fitness. According to Jenkins (1991), linear scaling improves the reproduction rates of better solution sets, while depressing the rates for the weaker solution sets.

A fitness proportionate selection scheme (sometimes called roulette wheel) is used to select individuals for the mating pool and for participation in reproduction. The number of copies of each individual selected for mating is based on scaled fitness using,

$$N_i = N_{total} \left(\frac{f'_i}{\sum f'} \right) \quad (2.25)$$

where, N_{total} is the total number of individuals in the population; f'_i is the scaled fitness and N_i is the number of copies of the i^{th} individual.

Two common crossover operations associated with binary representation are one point and uniform illustrated in Figure 2.4. It is important to emphasize the ability of binary representations to change the genes of the individuals (essentially mutating the genes) with the crossover operations. Uniform crossover randomly generates a mask (*i.e.* a string of “1s” and “0s” of the same length as the individual chromosomes) and the bits of randomly selected individuals are flipped according to the mask (*i.e.* “1” flips the allele and “0” does not flip the allele). Notice that for uniform crossover, the genes of the both offspring are different from that of their parent individuals, which is very likely to

occur unless the randomly generated mask contains an entire string of “0” or an entire string of “1”.

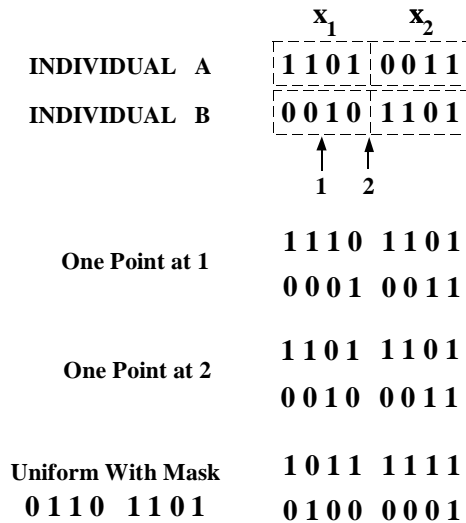


Figure 2.4: Crossover Operations for Binary String Representation

One-point crossover also offers the opportunity for genes to be changed by randomly selecting a crossover point and swapping the bit strings to the right of the crossover point of two randomly selected individuals according to a prescribed crossover rate (generally 50% to 75%). Notice that the one-point crossover operation at point 2 does not result in a change in the genes present in the parent individuals. The crossover of objects acts much like one-point crossover at point 2. It will be apparent in the next section that the crossover operation is the primary difference between binary string and object representations of the design variables, since object crossover is unable to produce new genetic material. As a result, object representation is more dependent of the mutation operation.

The mutation operation steps through the population and generates a random number between zero and one. If the number is less than prescribed mutation rate then the individual is mutated. Mutation simply selects an individual and changes one of the bits within the chromosome. The rate of mutation is generally low, in the range of 1% to 10% for binary representation.

Typically the optimal solution is not known and the evolution is carried out for a specified number of generations or until the fitness of the best individual does not change after a specified number of generations.

Jenkins (1991) used a one-point crossover operation at a rate of 1.0 and a mutation operation at a rate of 0.1 to obtain the optimal solution, $\mathbf{X} = [6 \ 2]$ after 14 generations and a population size of 10.

2.3.3 Evolutionary Algorithm with Object Representation

An object-oriented evolutionary algorithm (OO-EA) was written in the object-oriented programming language, Python (Python will be discussed in greater detail in Chapter 4). The source code for the OO-EA is provided in Appendix A. The OO-EA solves the simple example problem by implementing the same operations as the GA used by Jenkins (1991). Only the initialization and crossover operations are impacted by the difference in variable representation as discussed below.

Rather than assigning each individual a binary string of ones and zeros, each individual is assigned a solution set (*i.e.* a pair of attributes representing the values of x_1 and x_2). Figure 2.5 illustrates the heuristic tree representation for a population of ten

individuals. It should be noted that the individuals are objects that comprise the population.

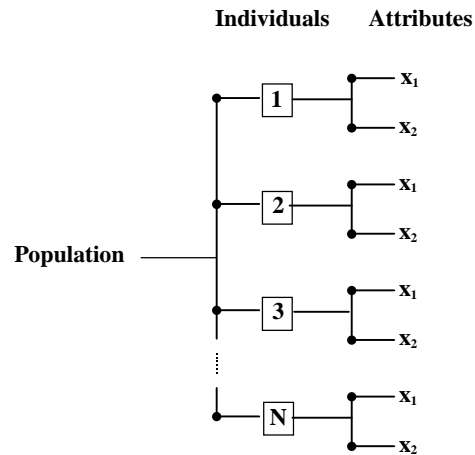


Figure 2.5: Population of Individuals with Object Representation

The crossover operation using an object representation occurs at the design variable level as shown in Figure 2.6. As previously discussed, the binary representation swaps at the allele level allowing the crossover operation to introduce a new genetic material (see Figure 2.4). Consequently, the object representation must rely on mutation since the object crossover operation cannot introduce new material. Furthermore, homologous and non-homologous crossover operations have been developed to offer more versatility to the object crossover operation illustrated in Figure 2.6. Homologous crossover can be seen to merely swap genes (*i.e.* design variables), while non-homologous crossover moves the genes along the chromosome.

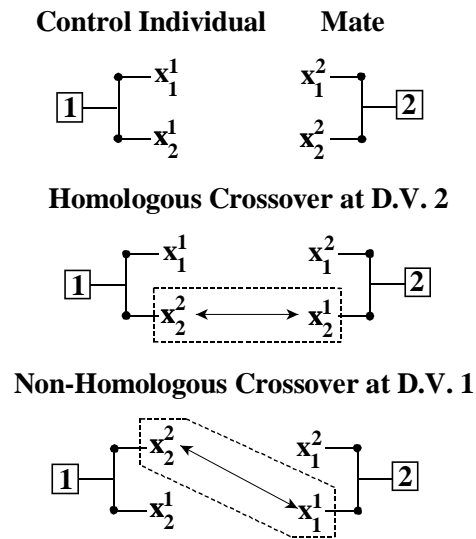


Figure 2.6: Crossover Operations for Object Representation

The Schemata Theorem (Goldberg 1989), has been used to assess the performance of GAs with binary string representations, however, it is not applicable to the object representation implemented by the OO-EA. Furthermore, research has been undertaken to elucidate the disruptive and beneficial nature of binary crossover and mutation in the evolutionary process (Wu et al. 1997). The object representation has the possibility to foster the creation and retention of beneficial building blocks. However, the object representation requires crossover and mutation be studied to ensure exploration of the design space. Voss and Foley (1999) studied the effect of non-homologous crossover on a simple cantilever optimization problem with a genetic representation (although hierarchical).

A parameter study was conducted to empirically explore the object representation and its ability to reliably find the solution that maximizes the previously discussed fitness function, (2.23). Linear scaling of the fitness values was used and fitness proportional

(roulette wheel) selection was employed Jenkins (1991). Success of the OO-EA is defined as finding the optimal solution. Design variables were restricted to integer values, $x_i \in \{1 \rightarrow 15\}$. The evolutionary parameters are defined as: the general crossover probability (p_c), subsequent probability of non-homologous crossover (p_{nh}), the probability of design variable mutation (p_m), and the population size (N_{indiv}). The effects of population size, mutation rate, and non-homologous crossover on the success of the OO-EA are given in Table 2.1, Table 2.2, and Table 2.3, respectively.

Table 2.1: Effect of Population Size on the Success of the OO-EA:
($p_c = 0.60$, $p_{nh} = 0.0$)

Population Size	Mutation Rate	Success Rate (%)
20	0.25	53
30	0.25	73
40	0.25	93
	0.30	93

Table 2.2: Effect of Mutation Rate on the Success of the OO-EA:
($p_c = 0.60$, $p_{nh} = 0.0$, $N_{indiv} = 20$)

Mutation Rate	Success Rate (%)
0.25	53
0.40	53
0.60	80
0.80	93
1.00	87

The success rate of the OO-EA appears to be linked to the population size and mutation rate. For small populations, the mutation rate should remain high so that sufficient new genetic material can be introduced in the population during the evolution.

If a small population is used, non-homologous crossover can improve the evolutionary search. However, the improvement is not as appreciable as mutation. When population sizes are sufficiently large (40 individuals in this case), the non-homologous crossover has less impact on the success rate and it may become detrimental at high rates.

Table 2.3: Effect of Non-homologous Rate on the Success of the OO-EA:
($p_c = 0.60$)

Population Size	Non-Homologous Crossover Rate	Success Rate (%)
$p_m = 0.25$	0.20	40
	0.40	40
	0.60	53
	0.80	67
	1.00	67
$p_m = 0.30$	0.20	93
	0.40	80
	0.60	93
	0.80	87
	1.00	60

Crossover operations are important to the OO-EA as with binary genetic algorithms. However, one should recognize that the non-homologous operator is important when more design variables are present and/or the hierarchical representation is “deep” (Voss and Foley 1999). This ensures that the crossover operations are able to move the genetic material around and explore building block formation.

2.4 Concluding Remarks

The focus of this thesis is to provide a vehicle to incorporate advanced analysis into the optimal design of unbraced steel frames. This chapter reviewed the available

optimization methods and included an example illustrating implementation of both gradient-based and direct search techniques.

The required gradient computations necessary for gradient-based techniques are relatively straightforward for linear elastic (geometric and material) behavior. However, gradients associated with the inelastic material and nonlinear connection behavior can be difficult to compute. Algorithms have emerged to iteratively solve nonlinear problems numerically using gradients of the continuously represented design variables. However, the number of possible constraints available in advance analysis based design (*e.g.* plastic hinge rotational restraints, connection rotation limits, etc.) coupled with a large number of complex design analyses can render gradient-based methods computationally prohibitive.

Researchers have developed methods to reduce the required iterative computational effort (*e.g.* optimality criteria and bound-and-branch methods). The optimality criteria (OC) method has been developed specifically for steel frame design, however, it requires that OC (based on corresponding sensitivity analysis) be written for each constraint in the problem. While this is relatively simple for deflection and allowable stress constraints, it can become prohibitively complex for performance based constraints (*e.g.* plastic hinge rotations). Furthermore, the virtual work and virtual strain energies used in previous research to develop the OC for stress constraints require linear elastic behavior, which is not viable when yielding within the members is considered. Thus, the gradient-based techniques require prohibitive effort for general application to complex analytical models such as that proposed in this thesis (advanced analysis based design).

Direct search techniques lend themselves to solving very complex optimization problems since they do not require continuous representation of the design variables and gradients of the objective and constraint functions. Although the direct search techniques require additional computational effort, it is a necessary evil to compensate for the (very high – maybe insurmountable) hurdles present in gradient-based techniques. The genetic and evolutionary algorithms are very good candidates to implement advanced analysis based design optimization. Furthermore, the proposed object representation shows promise to intuitively represent the member size and connection type design variables associated with the design of partially-restrained steel frames.

The subsequent chapters of the thesis formalize an advanced analysis optimization problem and offer a detailed description of the object-oriented evolutionary design algorithm developed to solve the optimization problem.

Chapter 3 – Advanced Analysis Based Optimization Problem

This chapter develops the optimization problem for the minimum weight design of unbraced steel frames. First, the methodologies and assumptions are briefly outlined for the advanced analysis method used to evaluate the frames. Second, the optimization problem is formulated to minimize the weight of the frame while satisfying serviceability, strength, and constructability constraints.

3.1 Methods of Advanced Analysis

The traditional steel design methods - Allowable Stress Design (ASD), Plastic Design (PD) and Load and Resistance Factor Design (LRFD) - are considered indirect methods since the stability of the frame and the stability of its members are considered separately when determining the ultimate strength of a frame. These methods require a first or second order analysis of the structural system and individual member capacity checks involving effective length factors and beam-column equations.

Methods of advanced analysis are direct methods since the stability of the structure and its members are considered as a whole resulting in a more accurate prediction of the behavior and ultimate strength of the frame (Chen and Seung-Eock, 1997). Furthermore, the use of advanced analysis methods to capture the strength and stability of a structure and its members liberates the designer from the need for tedious capacity checks for each member required by ASD, PD, and LRFD design specifications.

Several methods of advanced analysis have emerged over the past 30 years. These methods vary in ability to accurately capture the effects of yielding. The plastic-zone or distributed plasticity is known as an “exact solution” since it explicitly accounts for the second order effects, spread of plasticity, and residual stress. In fact, the (AISC 1993) beam-column equations were developed based on strength curves obtained from a plastic-zone analysis. However, it is computationally rigorous and has been deemed an academic tool with little practical use (Chen and Seung-Eock. 1997). The increasing processing speeds of desktop computers and the present research can be a catalyst for the future use of plastic-zone analysis in design offices.

The plastic hinge method greatly simplifies the advanced analysis by assuming the element remains elastic with zero length plastic hinges forming at the ends. Geometric non-linearity can be accounted for using one beam-column element per member and incorporating a stability functions. Although this method is far less computationally intensive with respect to the plastic-zone method, it does not account for the gradual spread of yielding at sections along the element or the effect of residual stresses. As a result, the refined plastic hinge method and quasi-plastic hinge method have been developed as variations to the plastic hinge method to improve analytical accuracy.

Of the previously mentioned advanced analyses, only the plastic-zone method qualifies as a method of advanced analysis according to (SSRC 1988). As defined, an advanced analysis is capable of considering residual stresses, imperfections, along the length and through the cross-section plastification, accurate tracing of the load-deformation response up to collapse, and ease in extension to three-dimensional analysis.

This section formulates the method of advanced analysis used to evaluate the performance of unbraced steel frames.

3.1.1 Inelastic Analysis Algorithm

The inelastic analysis algorithm used in this thesis employs a distributed plasticity or fiber element model. The spread of yielding throughout the cross-section of each beam and column member is modeled using sixty-six elemental fibers. Three fibers are used through the flange height with nine fibers across the flange width. Twelve fibers are used over the web height and are assumed to span the entire thickness of the web as shown in Figure 3.1-(a). Furthermore, residual stresses are assigned to each fiber prior to loading application along the length.

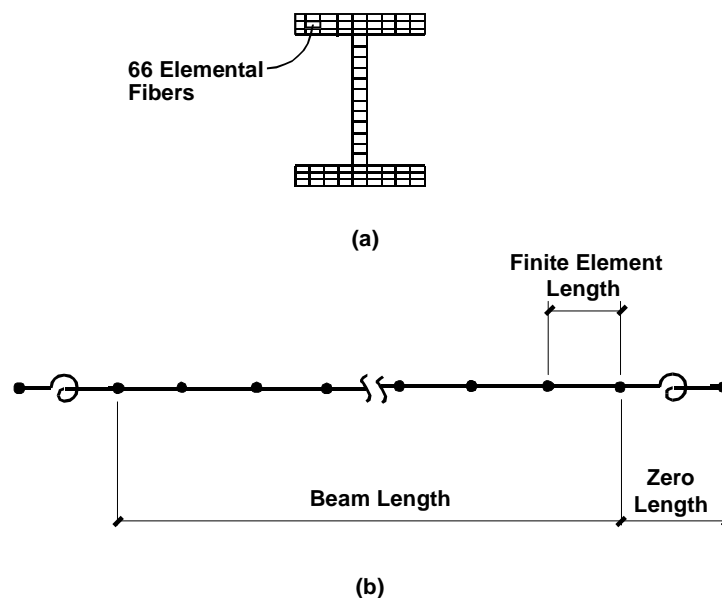


Figure 3.1: Fiber Element Model - (a) discretization of cross-section (b) discretization along member length

In addition, yielding is modeled using multiple finite elements along the length of each beam and column member shown in Figure 3.1-(b). The derivation of the finite element used to model beams and columns can be found in Foley (1996); Foley and Vinnakota (1999a); Foley and Vinnakota (1999b). In general, ten finite elements are required to provide accurate modeling of member behavior (Foley and Vinnakota 1997). The curvature within each finite element is an average value over its length.

The inelastic algorithm employs simple incremental stepping with load increments controlled by a requirement of constant work. While this technique can be limited in accuracy if initial load increments are not properly chosen (Foley and Vinnakota 1999a), it does not suffer from convergence failures normally associated with incremental-iterative analysis. The nonlinear load-deformation response is traced up to the frame collapse load or a user-defined applied load factor (whichever occurs first). The inelastic algorithm is terminated when any one of the following criteria are satisfied at the beginning of the next load increment:

- (a) the determinant of the structure stiffness matrix is less than or equal to zero.
- (b) the work done by reference loading for the current deformed configuration is less than or equal to zero.
- (c) the determinant of any member's stiffness matrix is less than or equal to zero.
- (d) the user-defined applied load factor is attained.

Other details of the inelastic algorithm can be found in Foley (1996) and Foley and Vinnakota (1999a).

3.1.2 Imperfections

Design specifications include member and frame imperfections in the equations used to assess member strength. Two types of imperfections should be considered in any advanced analysis design formulation: (a) member out-of-straightness; and (b) frame non-verticality (story out-of-plumb). Several excellent discussions of imperfections in steel frames are available (Bridge 1998; Bridge and Bizzanelli 1997; Maleck and White 1998). In this thesis, the concept of notional lateral loads is employed (Clarke and Bridge 1995; Hajjar 1997). This technique is a simple means with which to approximate story out-of-plumbness through assignment of “small” or “notional” lateral loads that are fractions of the gravity loads applied to each story in the frame. For example, the notional lateral loading at any story in the framework is (Clarke and Bridge 1995),

$$H_{\text{notional}} = 0.002P_{\text{story}} \quad (3.1)$$

where; P_{story} is the gravity loading (factored or service) at the story under consideration.

This notional lateral loading is factored using the lateral load factor and added to the factored lateral loading applied to the framework. Member out-of-straightness modeling is neglected since the present effort considers only unbraced frames. CEN (1993) and White and Nukala (1997) have developed criteria for which member out-of-straightness can be neglected. Maleck and White (1998) have reviewed these efforts and have concluded that member out-of-straightness can be omitted for typical unbraced frameworks.

3.1.3 Connection Model

Partially restrained connections are introduced into the inelastic model using zero length multi-linear spring elements attached to the ends of beam members. Connection characteristics follow the properties (sizes) of the connected beam using non-dimensional connection models (Bjorhovde et al. 1990). Past optimal design efforts for partially restrained frames utilized connections that were independent of the properties of the connected beam (Xu and Grierson 1993; Xu et al. 1995). Five non-dimensional connections intended to cover the range from fully restrained (FR) to flexible (pinned) were assumed in the present study. These connection curves are shown in Figure 3.2 with specific values found in Table 3.1. The connection moment is non-dimensionalized as,

$$\bar{m} = \frac{M_c}{M_{pb}} \quad (3.2)$$

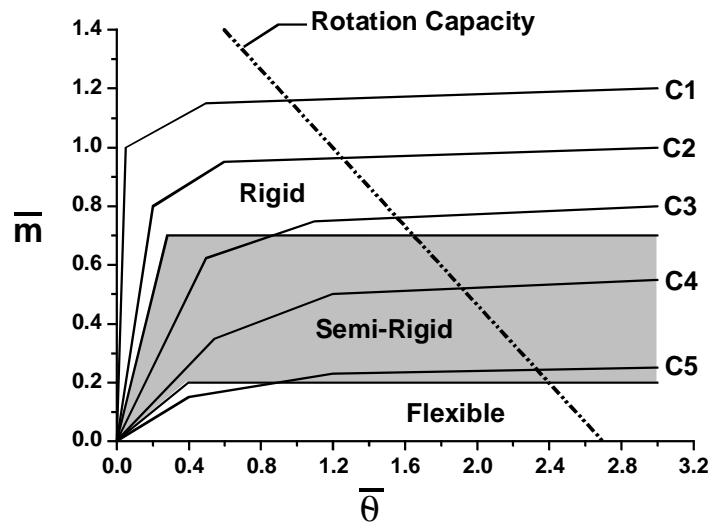
where, M_c is the connection moment and M_{pb} is the plastic moment capacity of the beam. The connection rotation is non-dimensionalized with respect to the beam flexural stiffness as,

$$\bar{\theta} = \frac{EI_b \theta_c}{M_{pb} (5d_b)} \quad (3.3)$$

where, θ_c is the connection rotation, d_b is the depth of the connecting beam, E is the modulus of elasticity of the connecting beam's material, and I_b is the second moment of area for the connecting beam.

Table 3.1: Multi-Linear Connection Modeling Data

Connection Mark	$\bar{\theta}$			\bar{m}		
	1	2	3	1	2	3
C1	0.050	0.500	3.000	1.000	1.150	1.200
C2	0.200	0.600	3.000	0.800	0.950	1.000
C3	0.500	1.100	3.000	0.625	0.750	0.800
C4	0.550	1.200	3.000	0.350	0.500	0.550
C5	0.400	1.200	3.000	0.150	0.230	0.250

**Figure 3.2:** Non-Dimensional Connection Curves

3.1.4 Local and Global Stability Considerations

The previously described inelastic analysis algorithm is unable to capture local buckling in the members, lateral-torsional instability of the members between brace points (columns), and out-of-plane flexural buckling of the columns. As a result, this section outlines three mechanisms used in the formation of the advanced analysis method to consider local and global stability.

3.1.4.1 Local Buckling

Omission of local buckling in the inelastic analysis is justified by ensuring that members in the candidate frameworks satisfy the web and flange slenderness provisions of the (AISC 1993) provisions for plastic design. Therefore, any column members in a framework must adhere to the following web slenderness limit,

$$\left(\frac{h}{t_w}\right)_{Limit} = \begin{cases} \frac{640}{\sqrt{F_y}} \cdot \left(1 - \frac{2.75 \cdot P_u}{\phi_b \cdot P_y}\right) & \text{for } \frac{P_u}{\phi_b \cdot P_y} \leq 0.125 \\ \frac{191}{\sqrt{F_y}} \cdot \left(2.33 - \frac{P_u}{\phi_b \cdot P_y}\right) \geq \frac{253}{\sqrt{F_y}} & \text{for } \frac{P_u}{\phi_b \cdot P_y} > 0.125 \end{cases} \quad (3.4)$$

where, P_u required axial strength, P_y is the yield strength, F_y is the yield stress of the steel, and ϕ_b is the resistance factor for flexure (assumed equal to one). Beams must satisfy the following web slenderness limit,

$$\left(\frac{h}{t_w}\right)_{Limit} = \frac{640}{\sqrt{F_y}} \quad (3.5)$$

The flanges in both beam and column members must satisfy,

$$\left(\frac{b_f}{2t_f}\right)_{Limit} = \frac{65}{\sqrt{F_y}} \quad (3.6)$$

3.1.4.2 Lateral-Torsional Buckling

Out-of-plane lateral-torsional buckling of members must also be prevented to ensure that the members in the framework have sufficient rotational capacity to allow formation of the frame's collapse mechanism(s). This is particularly important for the column members. Beams are assumed to have the top flanges continuously braced by the floor slab. The bottom flange of beam members is assumed braced at inflection points. As a

result, column members must satisfy the following unbraced length limit for plastic design, L_{pd} (AISC 1993),

$$L_{pd} = \frac{[3600 + 2200 \cdot (M_1/M_2)] \cdot r_y}{F_y} \quad (3.7)$$

where, M_1 is the smaller end moment, M_2 is the larger end moment, r_y is the radius of gyration about the minor axis., and F_y is the yield stress of the compression flange. The ratio of (M_1/M_2) is positive for moments causing reverse curvature and negative for single curvature.

3.1.4.3 Axial-Flexural Buckling

The out-of-plane axial buckling of column members is accounted by limiting the minor axis axial loading capacities to values of nominal axial strength, P_n computed using the inelastic buckling equations found in (AISC 1993),

$$P_n = \begin{cases} (0.658^{\lambda_c^2}) F_y & ; \lambda_c \leq 1.5 \\ \left(\frac{0.877}{\lambda_c^2} \right) F_y & ; \lambda_c > 1.5 \end{cases} \quad (3.8)$$

The slenderness parameter, λ_c , is based on a minor axis effective length factor equal to one.

3.2 Formation of the Optimization Problem

The most important considerations for steel frame design are safety, cost, and constructability. In most cases, cost is directly related to the weight of the frame; however, other factors including schedule deadlines, fabrication issues, construction site

accessibility, and availability of material can influence the economy of the design. The safety of the design is measured by strength and serviceability criteria. Factors that influence constructability are often project-specific. However, common design preferences include the repetition of member sizes, ensuring that columns “telescope” (*i.e.* larger and heavier columns are not stacked on top of smaller and lighter columns), and making sure that adjacent member shapes can physically be connected (*i.e.* the beam flange is not wider than the depth of the web of the connecting column). This chapter formulates the optimization problem used for the minimum weight design of FR and PR frames subject to service load, ultimate load, and constructability criteria.

3.2.1 Objective Function

The goal is to minimize the steel frame cost, which is generally directly related to its weight. Therefore, the objective function is written as the summation of column weights and modified beam weights using the following expression,

$$W = \sum_{k=1}^{N_{col}} L_k A_k \rho_k + \sum_{m=1}^{N_b} L_m A_m \rho_m \left[\left(\sum_{n=1}^{N_c} \zeta_n \right) - 1 \right] \quad (3.9)$$

where, ρ_k and ρ_m are the density of the column and beam material, respectively.

Furthermore, N_b , N_{col} , and N_c are the number of beams, columns, and beam connections, respectively. In all instances, the number of connections is two per beam.

The member lengths (column, L_k and beam, L_m) and cross-sectional area (column, A_k and beam, A_m) are dependent on the W-shape (*i.e.* design variable) of each member within the frame. The weight modification factor, ζ adjusts the beam weight to account for the cost of the connections at its ends. This factor is dependent on the connection type (*i.e.*

design variable), which increases as the connection stiffness and beam weight increases.

Table 3.2 shows typical values of these modifying factors for the beams. These modifiers are essentially those used by Xu et al. (1995) with some minor modifications resulting from the desire to have more connection possibilities.

Table 3.2: Connection Weight Modification Factors

Connection Mark	ζ_n		
	$A_m \rho_m \leq 35 \text{ plf}$	$35 \text{ plf} \leq A_m \rho_m < 161 \text{ plf}$	$A_m \rho_m \geq 161 \text{ plf}$
C1	1.550	1.700	1.850
C2	1.450	1.588	1.725
C3	1.350	1.475	1.600
C4	1.250	1.363	1.475
C5	1.150	1.250	1.350

3.2.2 Serviceability Criteria

The design must prevent excessive deflection, vibration, cracking, or any other behavior with potential to threaten the occupant's sense of security. The following constraints, ϕ_i , are used to restrict structural behavior of a frame at service load levels.

3.2.2.1 Attainment of Service Loading

Service load levels must be attained, therefore, the applied service load ratio, γ_s , must adhere to the following,

$$\phi_{\gamma_s} = \frac{1}{\gamma_s} \leq 1.0 \quad (3.10)$$

The service load combinations used in this thesis are those suggested by Ellingwood (1989),

$$\gamma_s [1.0 DL + 0.8 LL + 0.8 SL] \quad (3.11)$$

$$\gamma_s [1.0 DL + 0.4 LL + 0.4 SL + 0.5 WL] \quad (3.12)$$

$$\gamma_s [1.0 DL + 0.4 LL + 0.4 SL - 0.5 WL] \quad (3.13)$$

where, DL is the dead loading (including member self-weight), LL is the live loading, SL is the snow loading, and WL is the wind loading applied to the structure. The negative sign for the wind loading indicates loading applied from the left, while a positive sign indicates wind loading to the right.

3.2.2.2 Connection Rotation Limits

Connection rotations at service loads are limited to acceptable levels. Given the multi-linear moment-rotation response assumed for connections in the analytical model previously discussed, a natural rotational limit is the final connection rotation on the first linear branch of the model (see Figure 3.2). If rotations are less than this limit, the connection will theoretically return along its initial stiffness back to a state of zero set deformation. If rotation corresponding to the first stiffness is exceeded, the connection will theoretically unload along its initial stiffness and set deformation will result.

Therefore, at service load levels, the connection rotation, θ_s , must adhere to,

$$\phi_{\theta_s} = \frac{\theta_s}{\theta_{s,Limit}} \leq 1.0 \quad \xrightarrow{\text{with}} \quad \theta_{s,Limit} = \left[\frac{5 d_b M_{pb}}{EI_b} \right] \bar{\theta}_1 \quad (3.14)$$

where, M_{pb} is the plastic moment capacity of the beam member, E is the modulus of elasticity of steel, I_b is the moment of inertia of the beam, d_b is the depth of the beam, and $\bar{\theta}_1$ are the non-dimensional connection rotations (see Table 3.1).

3.2.2.3 Deflection Constraints

Inter-story drift and vertical deflections of the beams are important serviceability criteria to insure a sense of security of the occupants. Generally, specifications do not provide deflection limits. Therefore, a couple of widely used rules-of-thumb are used in this thesis. The inter-story drift, δ_H , must adhere to,

$$\phi_{\delta_H} = \frac{\delta_H}{\delta_{H,Limit}} \leq 1.0 \quad \xrightarrow{\text{with}} \quad \delta_{H,Limit} = \frac{h_{story}}{400} \quad (3.15)$$

where, h_{story} is the story height. Also, vertical deflection of the beams, δ_V , must satisfy,

$$\phi_{\delta_V} = \frac{\delta_V}{\delta_{V,Limit}} \leq 1.0 \quad \xrightarrow{\text{with}} \quad \delta_{V,Limit} = \frac{L_b}{360} \quad (3.16)$$

where, L_b is the beam span.

3.2.2.4 Yielding Constraints

The previously described advanced analysis considers geometric, material, and connection nonlinear behavior. The member cross-section must adhere to yielding limits at the service load level expressed as a percentage of the cross sectional area. This percentage is defined as,

$$\eta = \frac{A_e}{A}$$

where, A_e is the cross-sectional area of the remaining elastic core and A is the initial cross-sectional area prior to load application. The yielding of the cross-section must satisfy,

$$\phi_{\eta} = \frac{\eta}{\eta_{Limit}} \leq 1.0 \quad \xrightarrow{\text{with}} \quad \eta_{Limit} = 0.15 \quad (3.17)$$

In other words, the cross-section of a member shall not have more than 15% of its cross-section yielded under service loading. Other criteria have been proposed for plastic hinge based analysis, such as no plastic hinges shall form under service loads (White 1992; Ziemian et al. 1992a; Ziemian et al. 1992b). However, this thesis employs a distributed plasticity model that does not define plastic hinges – *per se*. A designer’s goal with respect to plastification within a structure should be to ensure that when the applied loads are removed from the structure, an acceptably low level of set deformation remains. The present 15% limit is assumed to correspond to an acceptably small level of probable set deformation upon removal of the service loads. The load level corresponding to the formation of a complete plastic hinge as assumed in other studies would (in general) be larger than the load level assumed in this thesis.

3.2.3 Strength Constraints

The frame must be capable of resisting the largest prescribed loads to ensure the safety of the occupants in extraordinary situations. The following constraints, ϕ_i , are used to restrict structural behavior of a frame under ultimate load conditions.

3.2.3.1 Attainment of Ultimate Loading

Buildings must attain target ultimate load levels. As a result, the applied ultimate load ratio, γ_U , must satisfy,

$$\phi_{\gamma_U} = \frac{1}{\gamma_U} \leq 1.0 \quad (3.18)$$

The ultimate load level combinations used in this Thesis are those suggested by (AISC 1993),

$$\gamma_U [1.2 DL + 1.6 LL + 0.5 SL] \quad (3.19)$$

$$\gamma_U [1.2 DL + 0.5 LL + 0.5 SL + 1.3 WL] \quad (3.20)$$

$$\gamma_U [1.2 DL + 0.5 LL + 0.5 SL - 1.3 WL] \quad (3.21)$$

3.2.3.2 Connection Rotation Constraints

Ultimate load levels can result in excessive demands on the connections in structural steel frameworks. Therefore, constraints on these rotations are assigned using limits proposed by Bjorhovde et al. (1990). Using the non-dimensional moments and rotations that define the moment-rotation response for the connection, a non-dimensional rotational capacity can be defined,

$$\bar{m} = 1.80 - 0.667\bar{\theta}$$

Once a beam member is chosen, the connection rotation at ultimate load levels, θ_U , must adhere to,

$$\phi_{\theta_U} = \frac{\theta_U}{\theta_{U,Limit}} \leq 1.0 \quad \xrightarrow{\text{with}} \quad \theta_{U,Limit} = 13.5 \left[\frac{d_b M_{pb}}{EI_b} \right] - 7.5 \left[\frac{d_b M_{pc}}{EI_b} \right] \quad (3.22)$$

where, M_{pc} is the plastic moment capacity of the connection.

3.2.3.3 Local and Member Instability Constraints

As loads approach ultimate levels, steel frames can undergo excessive deflection and material yielding with the potential to induce local and out-of-plane buckling of the

members. These instabilities are not captured by the distributed plasticity program incorporated in the method of advanced analysis and therefore, are addressed within the optimization problem via constraints based on the previously defined local buckling, lateral-torsional buckling, and axial (flexural) buckling limits.

The local buckling of the cross-section is addressed by constraining the width-thickness ratios of flanges and webs of each member. The flange width-thickness ratio, $b_f/2t_f$, must satisfy,

$$\phi_{b_f/2t_f} = \frac{b_f/2t_f}{(b_f/2t_f)_{Limit}} \leq 1.0 \quad (3.23)$$

where, the limit is expressed in (3.6). The web width-thickness ratio, h/t_w , must satisfy,

$$\phi_{h/t_w} = \frac{h/t_w}{(h/t_w)_{Limit}} \leq 1.0 \quad (3.24)$$

where, the limit for beams and columns is expressed in (3.5) and (3.4), respectively.

Lateral-torsional buckling is addressed by requiring the un-braced length, L , of the column members to satisfy,

$$\phi_{L_{pd}} = \frac{L}{L_{pd}} \leq 1.0 \quad (3.25)$$

where, L_{pd} is expressed in (3.7).

The axial buckling is addressed by requiring the ultimate axial force, P_u , within the column members to satisfy,

$$\phi_{P_n} = \frac{P_u}{P_{n, Limit}} \leq 1.0 \quad (3.26)$$

where, $P_{n, Limit}$ is expressed in (3.8).

3.2.3.4 Plastic Hinge Rotation Limits

The flange slenderness limits of beams and columns specified by (AISC 1993) assume that the plastic hinge rotation is defined by strain, ε , at the extreme fibers equal to $4\varepsilon_y$ (Yura et al. 1978). Another means to define plastic hinge rotational limits consistent with the local buckling criteria given in (AISC 1993) is to require the member curvature (plastic hinge rotation), κ , at ultimate load levels to satisfy,

$$\phi_{\kappa} = \frac{\kappa}{\kappa_{Limit}} \leq 1.0 \quad \xrightarrow{\text{with}} \quad \kappa_{Limit} = 4\kappa_y = 4 \frac{F_y}{E \cdot (d_b/2)} = \frac{8F_y d_b}{E} \quad (3.27)$$

where, κ_y is the curvature (pure bending) that corresponds to attainment of the material yield strain in the extreme tension and compression fibers for the pure bending condition.

3.2.4 Designer Preference Constraints

The optimization problem includes a shape constraint to ensure that columns “telescope”. The shape constraint requires that a column have the same shape or a larger shape than the column immediately above it. The AISC designation for a W-shape includes the nominal depth and weight per linear foot (*i.e.* a W8X31 has a nominal depth of 8 inches and a weight 31 pounds per linear foot). The shape constraint can be written as,

$$\phi_{shp} = \begin{cases} \frac{d_n^{upper}}{d_n^{lower}} \leq 1.0 \\ \text{or} \\ \frac{wt^{upper}}{wt^{lower}} \leq 1.0 \end{cases} ; \quad n = \text{number of columns} \quad (3.28)$$

where, d_n^{lower} and d_n^{upper} are the nominal depth of the lower and upper columns; wt^{lower}

and wt^{upper} are the weight of the lower and upper columns.

3.3 Concluding Remarks

The present chapter focused on the development of constraints used in the formulation of an optimization problem for steel frames designed using advanced analysis. The basics of an inelastic analysis algorithm employing distributed plasticity were described and criteria were outlined to ensure that a two-dimensional inelastic analysis is sufficient for the this research effort.

Chapter 4 outlines the method used to incorporate the previously defined objective function and constraints into an object-oriented evolutionary algorithm for automated steel frame design.

Chapter 4 - Evolutionary Design Algorithm

This chapter provides a detailed description of the evolutionary design algorithm developed to solve the advanced analysis based optimization problem formulated in Chapter 3. The algorithm is coded in the object-oriented (OO) programming language, Python and includes an “external” call to the operating system and an MS-DOS based program that carries out the advanced analysis described in Chapter 3. The goal of this chapter is to provide the necessary details to understand the proposed algorithm and the OO methodology used in its execution.

4.1 Python Terminology

The terminology specific to the object-oriented programming (OOP) language, Python, is an essential ingredient in providing a thorough description of the evolutionary design algorithm. This section provides a supplement to the brief introduction to OOP provided in Chapter 1 by defining terms and concepts specific to Python.

Python is a pure OOP language, in that everything is an *object*. The Python environment is based on the concept of *namespace* (referred to as *object space* in Chapter 1). Python *namespace* can be viewed as a place where names live – a pointer to a location in memory and it is divided into *modules* and *classes*. *Classes* produce multiple *objects* with similar behavior. There are two kinds of *objects* in Python: *class objects* and *instance objects*. The essentials of OOP in Python are summarized in the following (Lutz and Ascher 1999):

Class Objects Provide Default Behavior

- a.) The Python *class* statement generates a *class object* and assigns it a name.
- b.) Assignments within the *class* statement make *class attributes* (referred to as data in Chapter 1) accessible by *name qualification* (i.e. *object.name*)
- c.) Function statements (i.e. *def*) within a *class* generate *methods* (referred to as functionality in Chapter 1).

Instance Objects are Generated from Classes

- a.) When a *class object* is called, a new *instance* of an *object* is created.
- b.) Any *instance object* is given its own *namespace* (allocated memory) and inherits all *attributes* and *methods* found in the *class*.
- c.) When inside a *class method* function, reference to the *instance* of an *object* being processed can be made via the “self” *attribute*.
- d.) The `__init__` *method* can be thought of as a constructor *method*.

Python also supports the *abstraction*, *encapsulation*, and *polymorphism* concepts described in Chapter 1. Another extremely useful feature of Python is built-in types (which are also viewed as *objects* by Python). The basic built-in types are number and strings, which are common to most programming languages.

Another built-in type, “files” provides a way to create and gain access to “files” – named storage compartments residing within a computer and managed by the computer’s operating system – inside the Python environment. The proposed algorithm readily uses this built-in type to read “files” generated by the MS-DOS advanced analysis program.

Lists, tuples, and dictionaries are built-in data structures used extensively by the proposed algorithm. *Lists* and *tuples* are ordered collections accessed by offset (starting with an index of 0) and can contain arbitrary *objects* (i.e. numbers, strings, other *objects*). *Lists* and *tuples* can be searched, added to (*appended*), and sorted very efficiently using built-in functions. The only difference between *lists* and *tuples* is that *tuples* are immutable. In other words, the integrity of a *tuple* is sustained since *objects* contained in the *tuple* cannot be arbitrarily changed in place. *Dictionaries* are an un-ordered collection of *objects* in which values are accessed by key. *Objects* requiring frequent searching are best stored in *dictionaries*. It is important to emphasize that other programming languages like C or C++ require the programmer to establish these data structures by laying out data structures, managing memory allocation/deallocation, and writing sorting/searching routines. While in Python, these data structures are already established as built-in data types.

4.2 Algorithm Overview and Assumptions

Prior to a detailed description of the evolutionary design algorithm, this section provides an overview including underlining assumptions. The algorithm is a search procedure used to evolve optimized designs for steel frames with discrete design variables. Similar to the genetic algorithm (GA), the algorithm follows Darwin's survival of the fittest theory, in which an adaptive search is performed on a randomly generated population of solutions (Goldberg 1989; Holland 1975).

Unlike the GA, which can be applied to different types of problems, the proposed algorithm is specific to the design of steel frames. The GA can be implemented for

various types of optimization problems once the design variables are encoded into binary strings or other representations such as real value vectors and ordered lists (Spears et al. 1993). The proposed algorithm implements the same strategy as the GA by employing the following algorithmic procedure: (a) randomly generate an initial population of steel frames (initialization); (b) evaluate the “fitness” (*i.e.* quality) of the initial frames using a set of engineer-defined criteria (evaluation); (c) choose frames from past population based on fitness to form a mating pool (selection); and (d) combine or randomly alter the frames in the mating pool to form a new population of frames (reproduction).

The proposed algorithm generates a combination of member sizes and connection types for unbraced frames with known topology, loading, and material properties. An example of the required input to the proposed algorithm and evolved output (*e.g.* member sizes, W__X__, and connection types, C_) is presented in Figure 4.1. The member sizes are selected from a database of available AISC wide flanged sections and beam connections are selected from a pre-defined, non-dimensional list of connections ranging from rigid to flexible. The designer is left with the task of designing a connection with a moment-rotation ($\bar{m}-\bar{\theta}$) response compatible with the evolved connection (Foley and Vinnakota 1995; Kishi and Chen 1990).

4.3 Algorithm Details

This section provides a detailed description of the evolutionary design algorithm with extensive use of Python terminology. The proposed algorithm *encapsulates* the design process into seven *modules*: frame, aisc, cdata, initialization, evaluation, selection, and reproduction. The source listing for each module is provided in Appendix B.

The frame *module* contains the evolutionary algorithm used to drive the design process. A psuedo-code representation of the algorithm is outlined below and a flowchart of the algorithm is given in Figure 4.2. The algorithm steps are outlined below with detailed discussions of the *modules* taking place in later sections:

Step 1: The user-provided input includes:

- A *list* containing the frame topology, material properties, connection type, and design variable types (bldgInfo).
- A *list* of parameters for the inelastic analysis including: number of beam sub-elements, number of column sub-elements, load increment factor, and the number of load cases (evalInfo).
- A *list* of load criteria including: wind, floor live and dead loading, and roof live and dead loading (loadInfo).
- A *list* of constants used to define the lateral sway, vertical deflection, yielding of the cross-section, and member curvature limits (limitInfo).
- The number of individuals in the population (popSize).
- The generational parameters including the counter (genNum) and termination criteria (genMax).

Step 2: The initialization *module* establishes an initial population of randomly generated individual frames. The aisc and cdata *modules* are used to access databases containing AISC section properties and connection properties.

Step 3: The initial population is passed to the evaluate *module* where each individual frame is analyzed using the inelastic analysis program and violations of the constraints described in Chapter 3 are assigned numeric penalties.

- Step 4:** Fitness values for each individual in the initial population are calculated based on the weight and penalty(s) via a fitness *method*. In addition, “text files” are created to report the fitness and penalty values and to display the member shapes of each frame in the initial population via report and display *methods*.
- Step 5:** The selection *module* creates a mating pool by selecting individual frames based on individual fitness values.
- Step 6:** The reproduction *module* alters the individuals in the mating pool using crossover and mutation operations.
- Step 7:** The altered population is passed to the evaluate *module* where each individual frame is analyzed and violations of the constraints are assigned penalties.
- Step 8:** Fitness values of each individual in the altered population are calculated and the report and display “text files” are created.
- Step 9:** Steps 5–8 are repeated for the prescribed number of generations (genMax).

4.3.1 Initialization Module

The evolutionary process begins by establishing an initial population frames generated randomly based on the prescribed topology and material properties (modulus of elasticity, E and yield strength, F_y). The implementation of the random nature of the algorithm will be made apparent in subsequent paragraphs of this section. The topology parameters include the number of stories and bays (numStories and numBays), first and typical story heights (firStryHt and typStryHt), interior and exterior bay lengths (intBay and extBay), and bay width (bayWidth) (refer to Figure 4.1).

An object-oriented (OO) heuristic tree representation proposed by Voss and Foley (1999) is used to represent the population of frames. Each building *object* consists of story, column, beam, and connection *objects* as shown in Figure 4.3. The design variables are represented as “shape” *attributes* of the beam and column *objects* and “type” *attributes* of the connection *objects*.

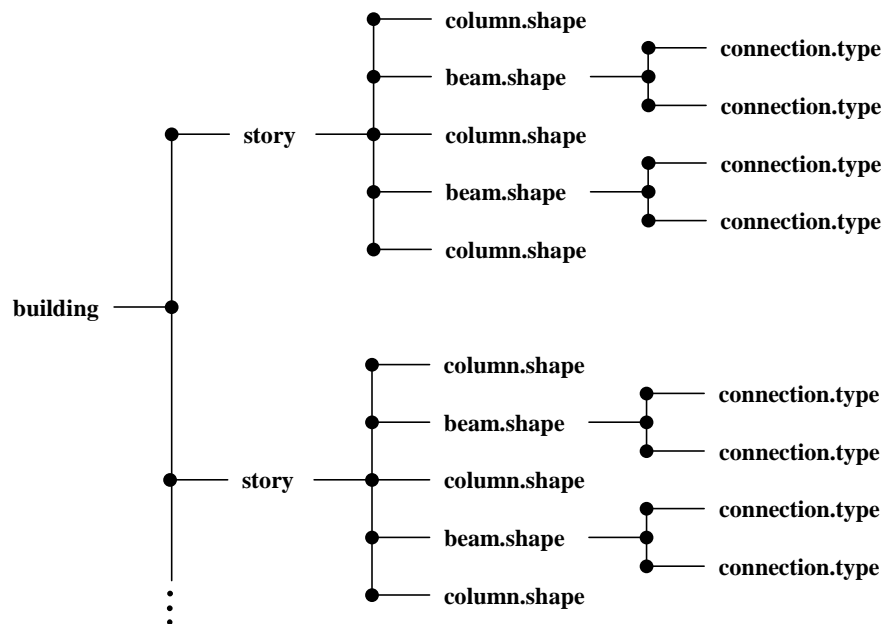


Figure 4.3: Object-Oriented Tree Representation of the Design Variables

The initialization *module* systematically constructs the initial population of frames using the hierarchy of classes in Figure 4.4. A population *object* is *instantiated* by calling the `__init__` *method* within the population *class* and assigning it a name, pop1. In addition to the topology and material property *attributes*, the population *object* is assigned empty *lists* to accommodate weight *attributes* (wt), fitness *attributes* (fit), and

building *objects*. The *dictionary* and *lists* associated with the selection of beam (dbBeams and bmList), column (dbColumns and colList), and connection *objects* (dbConn) will be discussed in subsequent paragraphs. The population *class* also contains *methods* necessary to calculate individual fitness values, report results, and display member shapes. It should be noted that the *attributes* and *methods* assigned to the population *object*, are accessed via *name qualification* (e.g. pop1.numBays, pop1.building, pop1.display). In other words, complete access to the parameters of the population can be transferred to other *modules*, *classes*, or *methods* by passing the single population *object* (pop1). This is one of the powerful aspects to object oriented programming that is lacking in traditional non-OO programming languages.

When a population *object* is *instantiated*, the prescribed number of building *objects* (popSize) are immediately constructed by passing the population *object* (pop1) to the `__init__` *method* and *appending* them into the building *list*. Each building *object* is assigned an empty story *list*. In addition, building *objects* acquire a *method* to calculate its weight, which is subsequently used by the fitness *method* in the population *class*.

The setStories *method* within the building *class* constructs the prescribed number of story *objects* (i.e. numStories) via the `__init__` *method* within the story *class* and *appends* them into the story *list*. Each story *object* is assigned an empty beam *list* and column *list*. The storyHt *attribute* assigned to each story *object* is a required parameter for subsequent use by the column *objects*. In addition, story *objects* acquire a *method* to calculate its weight that is used to determine the notional loads in the evaluate *module*. This is an example of *polymorphism* since the weight *method* is dependent on *name qualification*. In other words, the weight *method* assigned to the building *object* (i.e.

pop1.building[i].weight) calculates the weight of the i^{th} frame in the population.

Furthermore, the weight *method* assigned to story objects (i.e.

pop1.building[i].story[j].weight) calculates the weight of the j^{th} story of the i^{th} frame in the population.

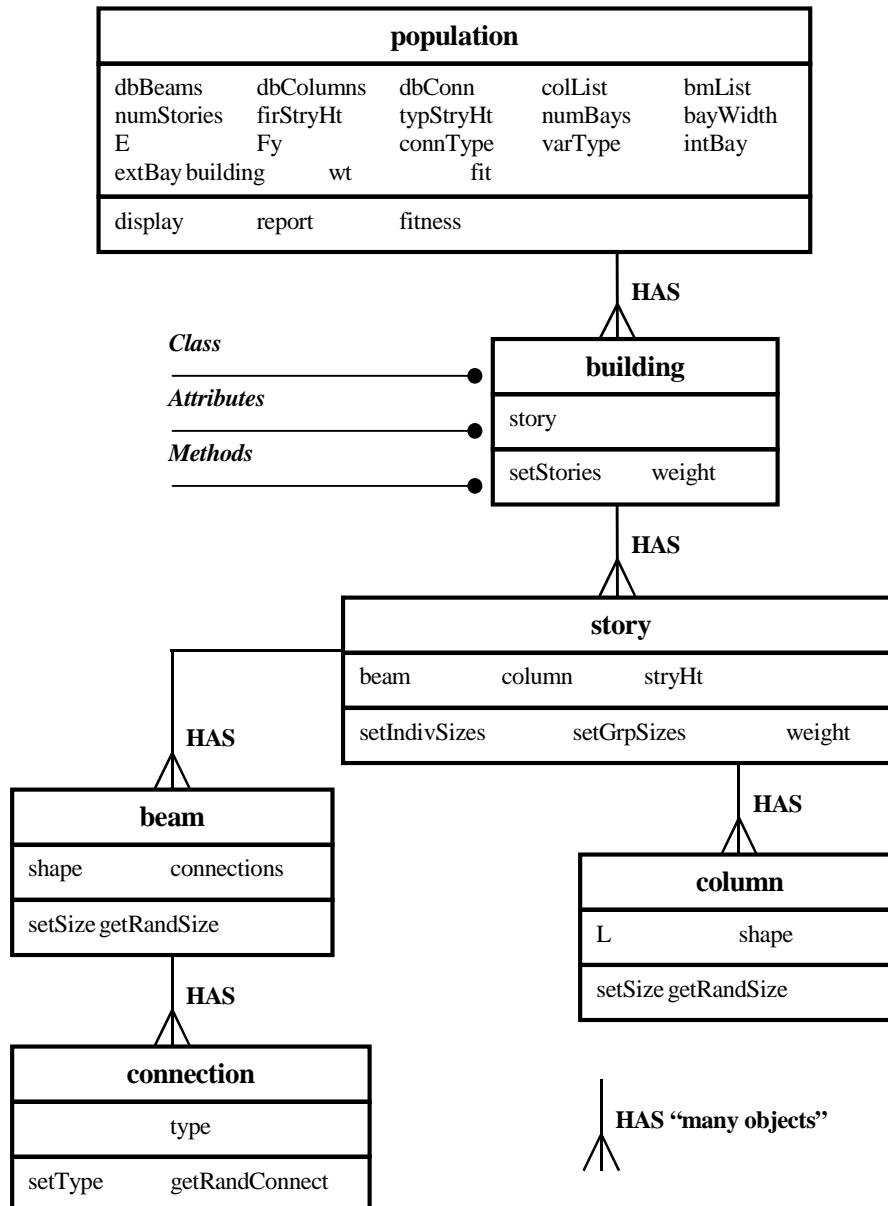


Figure 4.4: Initialization Module Class Hierarchy

The beam, column, and connection *objects* are constructed for the prescribed number of bays (`numBays`) by calling either the `setIndivSizes` or `setGrpSizes` *methods* within the *story class* depending of the variable type. The variable type parameter (`varType`) offers the option of individual or grouped design variables. The individual design variable option assigns *attributes* to each member independently. The grouped design variable option assembles the *objects* into groups and assigns the same *attribute* to each group. Figure 4.5 illustrates the difference between the two design variable types and the resulting number of variables. There are two advantages to grouping the design variables: (a) the design and construction of the frame becomes more practical since the number of member sizes and connection types is reduced resulting in a more manageable and constructable design; (b) the efficiency and consistency of finding an optimized design is improved since there are less combinations of member shapes and connection types resulting in a reduced solution space the algorithm must explore.

The population is “seeded” in order to reduce the initial solution space to include typical beam and column shapes found in the AISC Design Manual (AISC 1993). In other words, the beam and column *objects* are selected from a *list* of column W-shapes (`colList`) and a *list* of beam W-shapes (`bmList`) established via the *aisc module*. Beams are limited to W12, W14, W16, W18, W21, W24, W27, W30, W33, W36 shapes and the columns are limited to W8, W10, W12, and W14 shapes. This results in 149 available beam shapes and 71 available column shapes. Furthermore, the *aisc module* creates two databases or *dictionaries* containing section properties for both the column W-shapes (`dbColumns`) and beam W-shapes (`dbBeams`), which are accessed by keys representing the section properties (*i.e.* depth, radius of gyration, etc.).

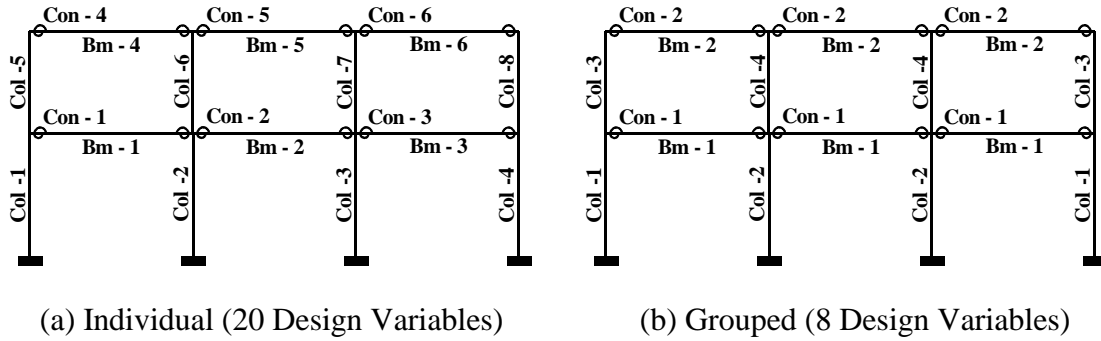


Figure 4.5: Design Variable Options

The column *objects* are constructed via the `__init__` *method* within the column *class* and are assigned length (L) and shape *attributes*. The distinction between the story height (`stryHt`) and length of column members (L) is necessary because of the subsequent inter-story crossover operations (*e.g.* non-homologous), which will be discussed in the in the reproduction *module*. The `setSize` *method* within the column *class* assigns each column *object* a shape *attribute* using the `getRandomSize` *method* and the `colList` *attribute* of the population *object* (`pop1`). The `getRandomSize` *method* provides the random nature of the algorithm by arbitrarily selecting a nominal depth from the `dbColumns` *dictionary*. A temporary *list* of columns is established from the previously generated `colList` based on the nominal depth and the flange slenderness constraint expressed in equation 3.23. It should be noted that the column web slenderness limit (equation 3.4) depends on the axial load and is addressed with a penalty in the evaluation *module*.

The beam *objects* are constructed via the `__init__` *method* within the beam *class* and are assigned an empty connection *list* and shape *attribute*. The `setSize` *method* within the beam *class* assigns each beam *object* a shape *attribute* using the `getRandomSize` *method* and the `bmList` *attribute* of the population *object* (`pop1`). The `getRandomSize`

method provides the random nature of the algorithm by arbitrarily selecting a nominal depth from the dbBeams *dictionary*. A temporary *list* of beam shapes is established from the previously generated bmList based on the nominal depth; the flange slenderness constraint 3.23 and the web slenderness constraint 3.24.

Connection *objects* are constructed via the `__init__` *method* within the connection *class* and are *appended* into the connection *list*, which is an *attribute* of the beam *object*. The setType *method* assigns type *attributes* by randomly assigning connections ranging from rigid (C1) to pin (C5). The getRandomConnect *method* arbitrarily selects a connection type by generating a random integer between 1 and 5. The connection database (dbConn) is created via the cdata *module* and contains values for the connection stiffness, rotation, and weight coefficients based on the connection model discussed in Chapter 3. The db.Conn *dictionary* is an *attribute* of the population *object* (pop1) and the connection information (stiffness, rotations) is keyed to the connection label (*e.g.* C1, C2, etc.). The connection type (connType) parameter allows the option of removing the connection stiffness as a design variable by setting all connections to a rigid (C1) condition.

Once the initial population of frames is established, the evaluation *module* analyzes and establishes the fitness of each frame.

4.3.2 Evaluation Module

Genetic and evolutionary algorithms require that the optimization problem be formulated as unconstrained. To this end, the weight computed using equation 3.9 is penalized to reflect violations of the problem constraints expressed in equations (3.10), (3.14) –

(3.18), (3.22)-(3.28). This penalized weight is defined as fitness. The unconstrained objective function or fitness is expressed as the product of the weight and penalties,

$$f = W \prod_{i=1}^{n_p} \Phi_i \quad (4.1)$$

where, W is the frame weight, Φ_i is the penalty corresponding to i^{th} constraint, and n_p is the number of constraints for the problem. This form of individual fitness is that used by Pezeshk et al. (1997) and Pezeshk et al. (2000).

The penalty multipliers, Φ_i , contained in (4.1) are formulated for each of the constraints considered in Chapter 3 by taking the product of the scaled constraint violations for each component and each load case. This is illustrated with the following generic expression,

$$\Phi_i = \prod_{r=1}^{N_r} \prod_{j=1}^{N_j} (p_i)_{j,r} \quad (4.2)$$

where, N_r is the total number of load cases, N_j is the total number of components (*e.g.* columns and/or beams and/or connections), and p_i is the scaled constraint violation associated with the i^{th} constraint, ϕ_i . The scaled constraint violations are established via the scaling functions proposed by Camp et al. (1996) and Camp et al. (1998). Linear and quadratic scaling functions are included in the proposed algorithm using,

$$p_i = 1.0 + k_i (q_i - 1)^n \quad (4.3)$$

where, n is the degree of the scaling function (1 for linear, 2 for quadratic), k_i is the scaling rate, and q_i is the scaling switch defined as,

$$q_i = \begin{cases} 1.0 & \text{if } \phi_i \leq 1.0 \\ \phi_i & \text{if } \phi_i > 1.0 \end{cases} \quad (4.4)$$

In other words, q_i ensures only constraints that are violated (*i.e.* $\phi = (\text{actual}/\text{limit}) \geq 1$) contribute to the penalty. Figure 4.6, demonstrates the effect of the scaling rate, k_i and the type of scaling function (linear or quadratic) on the scaled constraint violations.

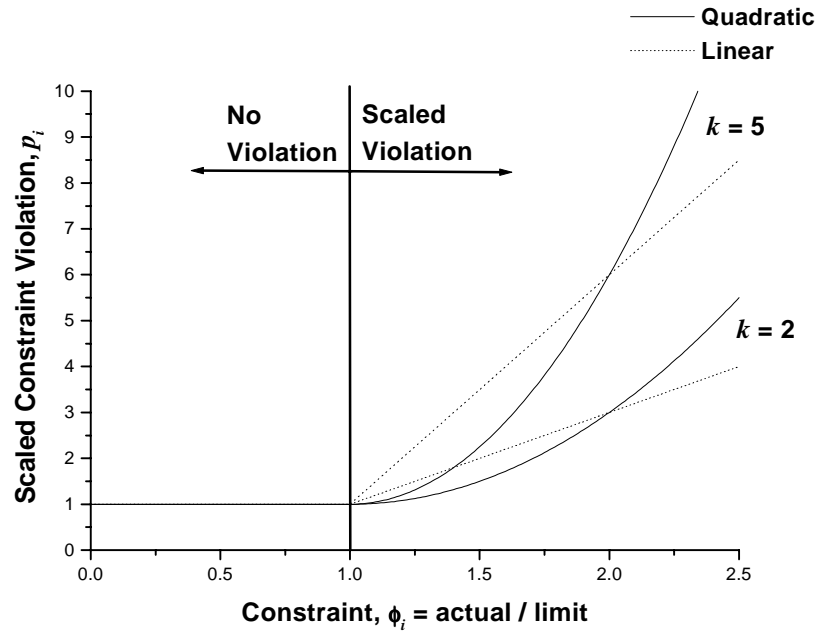


Figure 4.6: Scaling Functions

The expressions for penalty multipliers corresponding to each of constraints outlined in Chapter 3 are written below. The penalty multipliers for not attaining the service and ultimate load levels, respectively, are calculated using,

$$\Phi_{\gamma_S} = \prod_{r=1}^{N_S} (p_{\gamma_S})_r \quad (4.5)$$

and

$$\Phi_{\gamma_U} = \prod_{r=1}^{N_U} (p_{\gamma_U})_r \quad (4.6)$$

where, N_S and N_U are the number of service and factored load cases, respectively.

The penalty multipliers for connection rotations at service and ultimate load levels, respectively, are calculated using,

$$\Phi_{\theta_S} = \prod_{r=1}^{N_S} \prod_{m=1}^{N_b} \prod_{n=1}^{N_c} (p_{\theta_S})_{n,m,r} \quad (4.7)$$

and

$$\Phi_{\theta_U} = \prod_{r=1}^{N_U} \prod_{m=1}^{N_b} \prod_{n=1}^{N_c} (p_{\theta_U})_{n,m,r} \quad (4.8)$$

where, N_b is the number of beams in the framework and N_c is the number of connections at the ends of beams (two in all cases).

The penalty multipliers for inter-story drift and vertical deflection, respectively, are computed using,

$$\Phi_{\delta_H} = \prod_{r=1}^{N_S} \prod_{k=1}^{N_{col}} (p_{\delta_H})_{k,r} \quad (4.9)$$

and

$$\Phi_{\delta_V} = \prod_{r=1}^{N_S} \prod_{m=1}^{N_b} (p_{\delta_V})_{m,r} \quad (4.10)$$

where, N_{col} are the number of columns in the framework.

The penalty multipliers for excessive cross-sectional yield at service load levels and excessive curvature (plastic hinge rotation) at ultimate load levels, respectively, are calculated using,

$$\Phi_{\eta} = \prod_{r=1}^{N_S} \prod_{j=1}^{N_m} (p_{\eta})_{j,r} \quad (4.11)$$

and

$$\Phi_{\kappa} = \prod_{r=1}^{N_U} \prod_{j=1}^{N_m} (p_{\kappa})_{j,r} \quad (4.12)$$

where, N_m is the total number of members in the framework.

The penalty multiplier for column web local buckling, out-of-plane flexural buckling, and lateral-torsional buckling, respectively, are computed using,

$$\Phi_{h/t_w} = \prod_{r=1}^{N_U} \prod_{k=1}^{N_{col}} (p_{h/t_w})_{k,r} \quad (4.13)$$

and

$$\Phi_{P_n} = \prod_{r=1}^{N_U} \prod_{k=1}^{N_{col}} (p_{P_n})_{k,r} \quad (4.14)$$

and

$$\Phi_{L_{pd}} = \prod_{r=1}^{N_U} \prod_{k=1}^{N_{col}} (p_{L_{pd}})_{k,r} \quad (4.15)$$

The penalty multiplier for the designer preference criteria (shape) is computed using,

$$\Phi_{h/t_w} = \prod_{k=1}^{N_{col}} (p_{h/t_w})_k \quad (4.16)$$

The evaluation *module* contains a series of *methods* used to evaluate the performance of each frame based on the constraints outlined in Chapter 3 and the corresponding penalty multipliers calculated using equations (4.5) - (4.16). These *methods* can be categorized into the following two groups:

Analysis Methods:

advAnalysis
 getFileNames
 writeBuildingData
 writeStoryData
 writeBeamData
 writeColumnData
 writeConnectionData
 writeLatLoads
 writeGravLoads

Penalty Methods:

penalty
 calcPenalty
 connCurve
 weakAxisBuckling
 latTorBuckling
 webLocalBuckling

4.3.2.1 Analysis Methods

The analysis *methods* systematically generate the necessary input to run the advanced analysis (described in Chapter 3) for each frame in the population. The evaluation parameter *list* (evalInfo), load parameter *list* (loadInfo), and the population *object* (pop1) are the required input for the analysis *methods*. Since the evaluation *module* does not create *class objects*, the evaluation and load parameters cannot be accessed via *name qualification*. Therefore, in order for each *method* within the evaluation *module* to gain access to the evaluation (evalInfo) and load (loadInfo) parameters, they are declared *global variables* (available to all *methods* in the *module*).

The advAnalysis *method* sequentially assembles the required input for each frame under each load condition by passing an “input file” to each of the following series of *methods*.

The writeBuildingData *method* writes general material property and topology information including the specified modulus of elasticity and the number of members, nodes, supported nodes, and restraints to the “input file”. The nodes are numbered and

assigned coordinates. The members are numbered and the connecting nodes and member type are defined.

The `writeStoryData` *method* manages the assembly of the column, beam, and connection data. The “input file” is passed to the `writeColumnData` *method* and the necessary column information including: member number, flange width, flange thickness, depth, yield stress, and number of subelements is written. Once the data for each column is written, the beam and connection data is generated by passing the “input file” between the `writeBeamData` and `writeConnectionData` *methods*. The `writeBeamData` *method* writes the same necessary information for each beam member as was written for each column. The `writeConnectionData` *method* writes the stiffness and rotation data based on the connection model (presented in Chapter 3) for the connections at both ends of each beam.

The “input file” is passed back to the `writeBuildingData` *method* and the boundary conditions are applied by specifying degrees-of-freedom at each of the supported nodes (*e.g.* restrained or free). The loads are applied to the nodes by passing the “input file” to the `writeLatLoads` and `writeGravLoads` *methods*. The `writeLatLoads` *method* converts the wind and notional load to concentrated loads and applies them to the nodes along one of the exterior bays depending on direction the loads are applied (dictated by the load combination). The `writeGravLoads` *method* applies the distributed gravity and self-weight loading to the members of each story. Also, the self-weight of the columns is applied as concentrated loads at the nodes.

The `advAnalysis` *method* runs the advanced analysis generating an “output file” for each frame subjected to each load combination using the “output file” names

generated via the *getFileName method*. . A population of 50 frames subjected to the six load combinations (presented in Chapter 3), a total of 300 inelastic analyses are performed each generation. Clearly, the analysis methods occupy the majority of the computational time needed during the evolutionary design process.

4.3.2.2 Penalty Methods

The penalty *methods* read each “output file” generated by the advanced analysis program and returns penalty multipliers corresponding to the scaled constraint violations to the frame *module*. The limit criteria *list* (*limitInfo*), “output files”, and the population *object* (*pop1*), are the required input for the penalty *methods*. The penalty *method* establishes the constraints (*i.e.* $\phi_{\delta_H} = \delta_H / \delta_{H,Limit}$) applied to each frame in the population.

The majority of the constraints are generic and do not require knowledge of member properties. However, four constraint criteria involve limits that are dependent on member properties and/or results from the advanced analysis: connection rotations (at service and ultimate load levels), column axial buckling, column lateral-torsional buckling, and web local buckling of the column members.

The connection stiffness is dependent on the corresponding beam depth, plastic section modulus, and moment of inertia. These properties can be obtained with the *shape attribute* of the beam *object* and the database of beams (*dbBeams*). Also, the connection information can be accessed via the *type attribute* from the database of connections (*dbConn*). Therefore, the connection rotation limit is determined by passing the beam *object* to the *connCurve method*.

The axial load limit $P_{n,Limit}$ is dependent on properties of the column members. Therefore, the `weakAxisBuckling` *method* requires the column *object* in order to gain access to its length, radius of gyration, and cross-sectional area. The length is an *attribute* of each column *object* and the radius of gyration and cross-section area are available in the column database (`dbColumns`).

The column length limit based on plastic design is dependent on the radius of gyration and the moments at each end of the column calculated in the advanced analysis. (recall equation 3.7). Therefore, the column *object* and end moments (read from the “output file”) are passed to the `latTorBuckling` *method*.

The column web-slenderness limit is dependent on the cross-sectional area and the axial load under ultimate load conditions. Therefore, the column *object* and the axial load (read from the “output file”) are passed to the `webLocalBuckling` *method*.

Each constraint is passed to the `calcPenalty` *method* and scaled constraint violations are calculated using the scaling functions formulated in the previous section. As previously noted, the constraints *not* in violation are assigned a scaled constraint violation of one, and therefore, have no contribution to the fitness function. The penalty multipliers corresponding to each of the constraints are *appended* into a *tuple* containing the product of all the scaled constraint violations associated with each constraint and the corresponding individual frame. The penalty multipliers are then *appended* into the *penalty list*, which is an *attribute* of the population *object* (`pop1`), and returned to the *frame module* and subsequently used by the *fitness method*.

Once the individual fitness of each frame has been established, the *selection module* forms a mating pool of frames chosen based on fitness.

4.3.3 Selection Module

Selection is vital to the performance of the genetic and evolutionary algorithm. An effective evolutionary algorithm includes a bias to select superior or highly fit individuals in later generations while maintaining a diverse population for early generations. This diversity is provided by less fit individuals that still possess “good” genetic material (*i.e.* design variables). Two popular methods include roulette-wheel (fitness proportionate) selection and a form of tournament selection that includes a bias for selecting individuals from two partitions of the population.

The selection *module* contains a roulette (wheel) *class* and tournament *class* shown in Figure 4.7. In addition, the `saveBestIndiv` *method* assigns the fittest frame (*i.e.* building *object*) as an *attribute* of the population (`pop.bestIndiv`) in order to provide the option to incorporate elitism in the subsequent reproduction *module*.

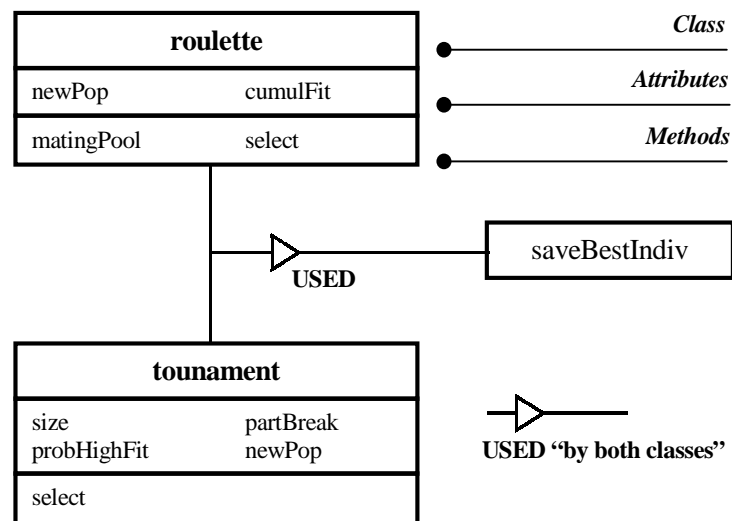


Figure 4.7: Selection Module Class and Method Hierarchy

4.3.3.1 Roulette Wheel Selection

Fitness proportionate selection techniques assign individuals a probability of being chosen that increases with fitness. The selection scheme used in the illustrative example in Chapter 2 is an example of fitness proportionate selection. A physical analogy for this technique can be visualized by allocating fractions or percentages of the fitness of the population as whole to each individual. The percentages can be visualized as segments of a line, pieces of a pie, or slots in a roulette wheel.

The roulette *class* is based on a procedure contained in (Rajan 2001). A roulette *object* is *instantiated* and assigned an empty *list* (*newPop*) that will temporarily house the selected individuals. The *matingPool method* maps the individuals of the population to segments of the roulette wheel based on a scaled fitness as shown in Figure 4.8. If the perimeter of the roulette wheel is one, then that portion of the perimeter corresponding to each segment represents a percentage of the wheel. In other words, the larger the percentage of the wheel an individual represents, the greater the probability of being selected. Each individual and its resulting percentage of the wheel are *appended* into a *tuple* (*cumulFit*), which is passed to the *select method*.

The *select method* generates a random number between zero and one and selects the individual corresponding to the range (*i.e.* segment) in which the random number falls. The selected individuals are *appended* into the temporary *newPop list*. Once the prescribed number of individuals are selected (*popSize*), the building *list* (*pop1.building*) is cleared and the selected individuals are *appended* into the building *list*, thus creating a new population.

p_i is the probability of selecting the i^{th} individual
 n is the population size

Selection Steps:

1. generate random number p between 0 and 1.0
2. if $p_i > p$ then select the i^{th} individual into the mating pool

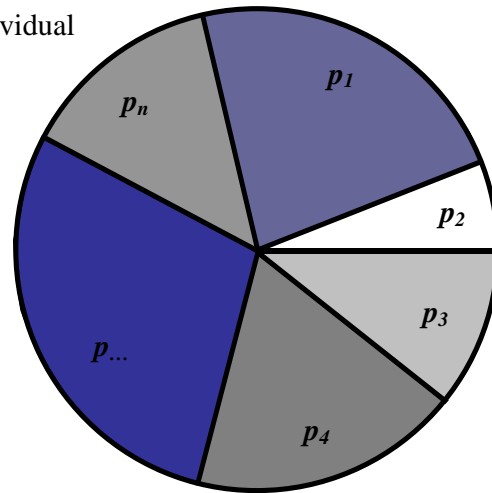


Figure 4.8: Roulette Wheel Selection Scheme

Although, the roulette-wheel technique is intuitive, it offers no means of regulating the selection pressure (bias towards selecting “better” individuals). As a result, Whitley (1989) proposed that fitness proportionate selection schemes can impede the progress of the EA by either stagnating the evolution due to lack of selection pressure or prematurely terminating the evolution by converging too quickly.

4.3.3.2 Tournament Selection

Tournament selection promotes selection pressure by organizing tournaments and selected individuals compete based on fitness. The size of the tournament can range from two individuals to the total number of individuals in the population (popSize).

The tournament *class* selects the competing individuals by partitioning the population via a group selection scheme (Camp et al. 1996; Camp et al. 1998; Pezeshk et al. 1997; Pezeshk et al. 2000). A tournament *object* is *instantiated* and assigned an empty *list* (newPop) that will temporarily house the selected individuals. The *select method*

sorts the population of individuals based on fitness and segregates the individuals into two groups based on the prescribed partition parameter (`partBreak`). A second parameter (`probHighFit`) specifies the probability of selecting from a particular partition as shown in Figure 4.9. A random number is generated and if the number is less than the `partbreak` parameter an individual is selected randomly from the partition with the fitter individuals. Otherwise, an individual is selected randomly from the partition containing less fit individuals. The total number of tournaments is equal to the size of the population (`popSize`). The winners represent the fittest individuals within each tournament and are subsequently *appended* into the temporary *newPop list*. The *building list* (`pop.building`) is then cleared and the selected individuals are *appended* into the *building list*.

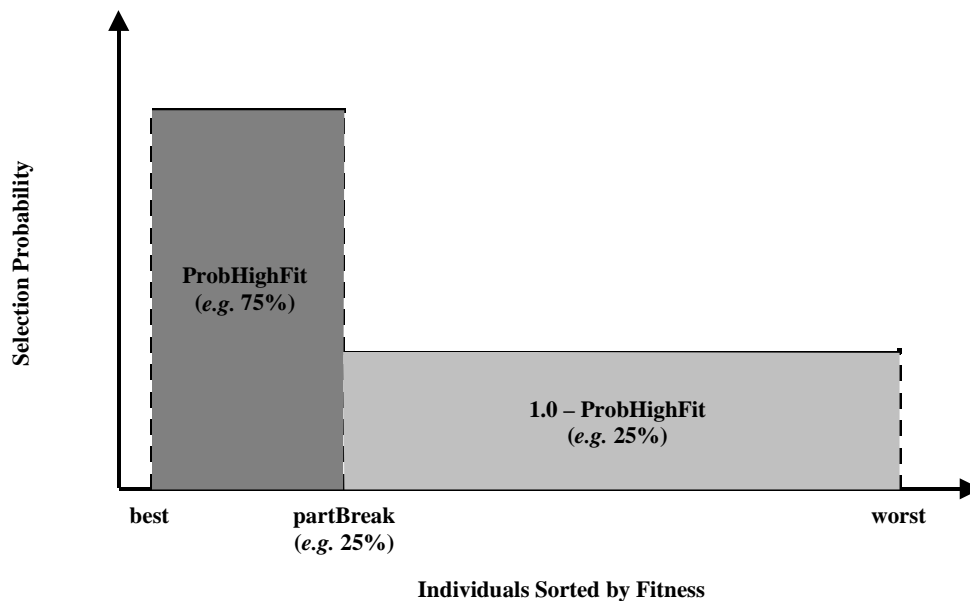


Figure 4.9: Group Selection (Partitioning) Scheme

This *method* offers considerable control of the selection pressure. For example, individuals can be selected from the top 25% of the population at a rate of 75%, leaving only a 25% chance of selecting from the less fit individuals. These rates can vary with generation if desired. As a result, egalitarian selection can be included earlier in the evolution while a high bias toward more fit individuals can be introduced later in the evolution. In a study of selection techniques, Yang and Soh (1997) concluded that tournament selection out performed the roulette-wheel selection in enhancing the efficiency of the search.

Once the mating pool has been formed, the reproduction *module* recombines the frame components and forms a new population of frames that is passed on to the next generation.

4.3.4 Reproduction Module

The natural phenomenon of reproduction is simulated using crossover and mutation operations. The crossover operation exchanges the genetic material of two parent individuals to create an offspring. Mutation adds diversity to the population by randomly introducing new genetic material. The extent of crossover and mutation is regulated via prescribed rates. Although, there is not an established standard or rule-of-thumb defining crossover and mutation rates, it is common to use higher crossover rates (*i.e.* 0.25 to 0.30) and lower mutation rates (*i.e.* 0.01 to 0.10). Ultimately, it is left to the user's discretion and the limited parameter study in Chapter 2 demonstrates the need for a higher mutation rate for an object representation of the design variables.

The reproduction *module* contains the crossover *class* and mutation *class* shown in Figure 4.10. In addition, the *module* offers the optional elitism *method*. Elitism ensures the fittest frame bypasses the reproduction operation and is carried through to the next generation. The fittest frame replaces the least fit frame in the subsequent generation.

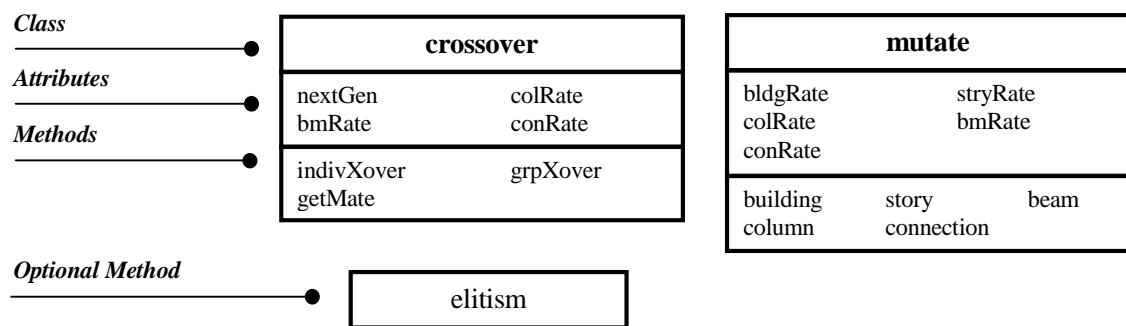


Figure 4.10: Reproduction Module Class and Method Hierarchy

4.3.4.1 Crossover

The *object* representation of the design variables used by the proposed algorithm does not lend itself to the traditional crossover methods used by genetic algorithms with a binary representation of the design variables as discussed in Chapter 2. The proposed algorithm implements a crossover scheme intended to simulate the uniform crossover technique used with binary representation of the design variables (Camp et al. 1996; Camp et al. 1998; Hayalioglu 2000; Rajan 2001). This scheme uses the homologous and non-homologous crossover operations introduced in Chapter 2. To illustrate these operations, consider the frames in Figure 4.11. In the case of homologous crossover, a roof beam in the exterior bay of one frame can be swapped (exchanged) with a roof beam in the

exterior bay of a second frame. The non-homologous crossover offers the ability to exchange objects between different stories and bays. For example, an interior column at the first story of one frame can be swapped (exchanged) with an exterior column at the third story of a second frame.

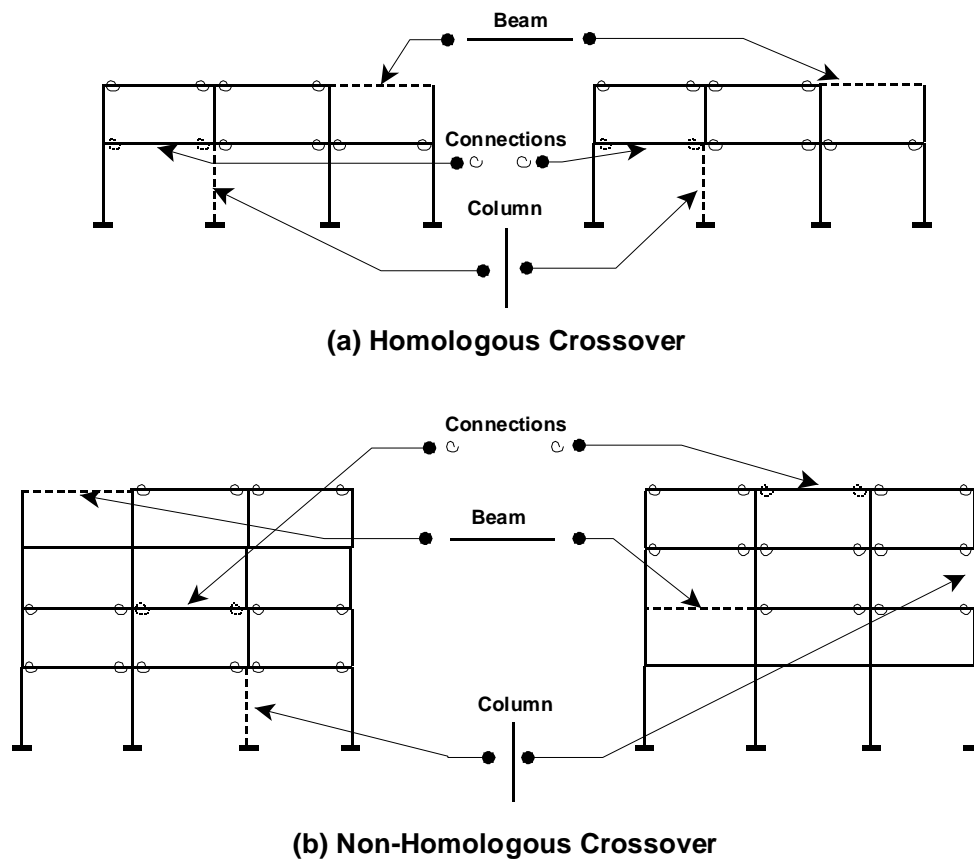


Figure 4.11: Crossover Operations

The crossover *object* is *instantiated* and assigned an empty *list* (nextGen) to temporarily house the next generation of frames. One of two *methods* is used to perform the crossover operation depending on the variable type (varType). The indivXover

method is used for individual variables and the *grpXover method* is used for grouped variables. Both *methods* systematically step through the mating pool of frames and swap the shape *attributes* (*i.e.* design variables) of the column and beam *objects* and the type *attributes* (*i.e.* design variables) of the connection *objects*.

Each frame is sequentially defined as the control frame for the crossover operation. The *getMate method* randomly selects a second frame or mate from the mating pool ensuring it is not the same as the control frame. The function of the mate is to provide material to be placed within the control frame. Each column, beam, and connection *object* of the control frame is given the opportunity to be swapped depending on the prescribed rates of column (*colRate*), beam (*bmRate*), and connection (*conRate*) crossover. A random number is generated for each column, beam, and connection *object*. The design variable *attributes* of similar *objects* are swapped (*i.e.* columns swap with columns) only if the random number is less than the corresponding crossover rate. Furthermore, there is an equal probability the *attributes* will be swapped in a homologous or non-homologous manner. The control frame represents the offspring and is subsequently *appended* into the next generation *list* (*nextGen*). Once the crossover operation is performed on each individual in the mating pool, the building *list* is cleared and the *nextGen list* is *appended* into the building *list*. At this stage, generation of the new population is complete.

4.3.4.2 Mutation

The mutation *class* randomly selects and assigns new *attributes* to *objects* at each level of the hierarchy of the population (*i.e.* building, story, column, beam, and connection) in

order to maintain diversity within the evolutionary algorithm and explore outlying areas of the design space.

The *mutate class* contains building, story, column, beam, and connection *methods* used to randomly alter each of the five primary objects of the population hierarchy. The extent of mutation is controlled by specified rates (*bldgRate*, *stryRate*, *colRate*, *bmRate*, and *conRate*). The *methods* randomly select a predefined number (equal to the size of the population *popSize*) of each of the primary *objects* and generate new design variable *attributes* according to the prescribed mutation rates. For example, the column *method* arbitrarily selects a frame from the population and then arbitrarily selects a column *object* from the selected frame. A random number is generated between 0 and 1 and if the number is less than the prescribed rate of column mutation, a new shape *attribute* is generated otherwise the *attribute* remains the same. The new *attribute* is assigned simply by calling the corresponding *method* used to establish the initial population within the initialization *module*. Recall that once a column *object* was *instantiated* in the initialization *module*, it was given the ability to establish its shape *attribute* via the *setSize method*. Similarly, a building *object* can regenerate itself via the *setStories method* (thus creating an entire new frame). A story *object* can regenerate itself via the *setIndivSizes* or *setGrpSizes methods* (depending on the design variable type). A beam *object* can regenerate itself via the *setSize method* (note the *polymorphism*) and a connection *object* can regenerate itself via the *setType method*.

Once the next generation of frames is established, the next step is to return to the selection *module* and repeat the evolutionary process until the prescribed number of

generations is achieved. After the final generation, the “optimized” design corresponds to the fittest frame (*i.e.* lowest modified weight).

4.4 Concluding Remarks

This chapter provided a detailed depiction of the development of the evolutionary design algorithm with profound emphasis on object-oriented programming (OOP) aspects. The focus of this chapter is to offer the reader an understanding of the underlying concepts of the proposed algorithm, while providing the specific details of the OO implementation for readers with an interest and savvy in OOP.

The chapter began with a review of OOP terminology specific to the Python language followed by an introduction to the evolutionary design algorithm including the basic assumptions. The majority of the chapter describes the primary components of the evolutionary process: initialization, evaluation, selection, and reproduction. Initialization is the process of randomly generating a population of frames from a group of available member sizes and connection types. The population is evaluated using an inelastic analysis program and prescribed constraint criteria to establish a fitness for each frame in the population. Frames are selected into a mating pool based on fitness. The members and connections of the frames in the mating pool are exchanged and altered using crossover and mutation operations resulting in a new population carried over to the next generation. The evolution continues for a prescribed number of generations.

Trial design runs using the proposed algorithm are presented in Chapter 5. Three frames found in the literature are used to measure the performance of the proposed algorithm.

Chapter 5 – Frame Design Examples

This chapter presents the application and evaluation of the evolutionary design algorithm developed in this thesis. The primary objectives are to demonstrate the ability of the proposed algorithm to obtain practical designs and evaluate the algorithm's performance. Assessment will be carried out through brief discussions and observations of the results. Furthermore, the results will be used in Chapter 6 to offer improvements and suggestions for future research.

Three studies were conducted using frames found in the literature. A three-story, two-bay frame (Barakat and Chen 1990) and a two-story, three-bay frame (Xu et al. 1995) were used to assess the applicability of the proposed algorithm to moderately sized frames. A ten-story, three-bay frame (Xu and Grierson 1993) was used to illustrate the scalability of the proposed design algorithm.

5.1 Frame 1 - Three-Story, Two-Bay

The first frame considered is the three-story, two-bay frame analyzed by Barakat and Chen (1990). The topology and loading of the frame is shown in Figure 5.1. The frames are spaced at 20 feet on center with a wind pressure of 33.7 psf. The steel is assumed to have a modulus of elasticity of 29,000 ksi and yield strength of 36 ksi. Additional design parameters are found in Table 5.1.

The frame was designed for two conditions: (a) FR connections and (b) PR connections. Three design runs were conducted for each condition to assess the consistency and repeatability of the results. The performance of the proposed algorithm

and structural behavior of the fittest frames for both connection types will be investigated via fitness trajectories, load response curves, and penalty convergence plots.

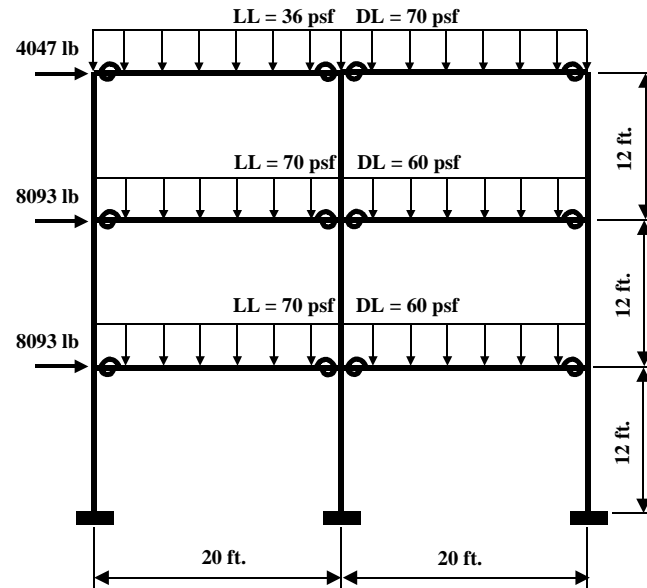


Figure 5.1: Topology and Loading for Frame 1

Table 5.1: Design Parameters for Frame 1

Evolutionary Parameters:

Grouped Design Variables	9 (FR), 12 (PR)
Population Size	50
Generations	50

Evaluation Parameters:

Sub FE - columns	10
Sub FE - beams	10
Initial Load Increment	0.05

Reproduction Parameters:

	Crossover	Mutation
Building Rate	na	0.05
Story Rate	na	0.05
Column Rate	0.75	0.05
Beam Rate	0.75	0.05
Connection Rate*	0.75	0.05

Selection Parameters:

Tournament Size	2
Partition Break	0.4
Higher Fitness Probability	0.7
Elitism	on
Penalty Function	linear
Penalty Rate	2

* Suppressed for FR connection runs

The convergence of the proposed algorithm was measured by plotting fitness trajectories as shown in Figure 5.2. The trajectories generationally report the fitness (*i.e.* the modified weight accounting for connections and constraint violations) for the three runs performed for both conditions (a) and (b). As expected, the trajectories start with heavier designs and ultimately converge to lighter designs. However, the stagnation occurring at relatively early stages in the evolutionary process (within 5 to 10 generations) is noteworthy. This illustrates the ability of the proposed algorithm to recognize and exploit the good “genetic” material contained within the population of frames. However, this also exposes a tendency of the proposed algorithm to focus on portions of the solution space. This is a common characteristic of stochastic algorithms. In these algorithms, solutions tend to reside within the landscape of the solution space provided by the initial random generation of solutions and subsequent mutation. Consequently, there is no guarantee the proposed algorithm will converge to the single, “optimal” solution (*i.e.* minimum weight design). However, the solutions are seen to be “good” from an engineering perspective.

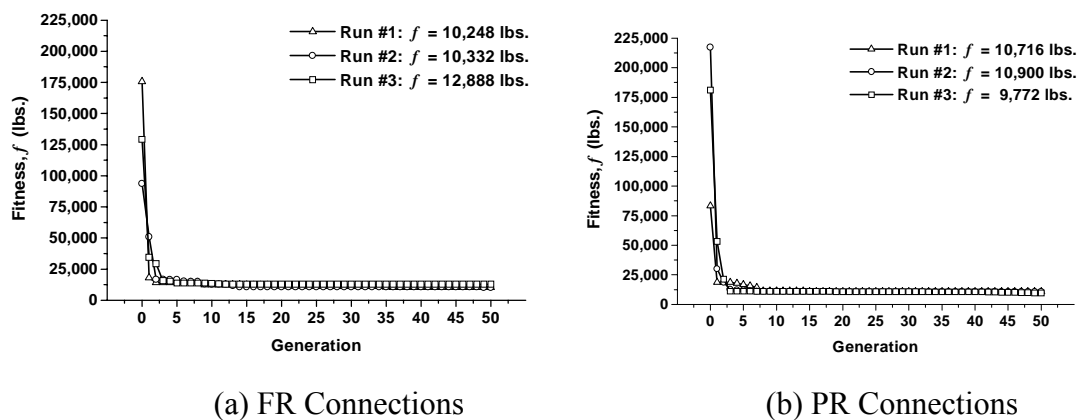


Figure 5.2: Fitness Trajectories for Frame 1

It is important to observe that within each set of runs no two evolved frames are the same, which is also common with stochastic search procedures (Kameshki and Saka 1999, Pezeshk et al. 2000, Camp et al. 1998). This can be attributed to the random nature of the evolutionary algorithm as it searches portions of the solution space. It is often impractical to exhaustively search the entire solution space for the very large number of combinations of member sizes and connections types possible for a given frame. The design of steel frames is an ideal candidate for stochastic methods since the solution space is fraught with feasible designs within a narrow range of minimum weight (*i.e.* local minimum). This can be seen in the evolved configurations for both frames with FR and PR connections in Figures 5.3 and 5.4. The results can be viewed as “optimized” designs, which the designer can further evaluate and massage into a final design.

The majority of the computational effort in the proposed algorithm is found in the evaluation of the frames. Six (one for each load case) distributed plasticity analyses were conducted for each frame, each generation. The current study considers a population of 50 frames over 50 generations, resulting in 15,000 distributed plasticity analyses for each run. All runs were made on a Pentium-based personal computer with a 400 MHz processor speed and run times for each evolutionary design were approximately six hours (wall clock).

The evaluation parameters (number of column sub-finite elements, $N_{fe,col}$; number of beam sub-finite elements, $N_{fe,bm}$; and initial load increment, $\Delta\delta$) have a significant impact not only on the accuracy of the analysis, but also on the duration of the evolutionary design process. In general, the evaluation parameters provided in Table 5.1 are adequate for providing accurate modeling of member and frame behavior (Foley

1996, Foley and Vinnakota 1999). However, to confirm the accuracy of the proposed algorithm, Figure 5.5 presents a comparison of the ultimate load response of the horizontal roof displacement under various inelastic analysis parameters. The ultimate load responses are plotted for the three ultimate load cases expressed in Chapter 3. The ultimate gravity load case, equation (3.19) is denoted by F1 and the ultimate lateral load cases, equations (3.20) and (3.21) are represented by F2 and F3. This terminology will be used for all subsequent load-response plots.

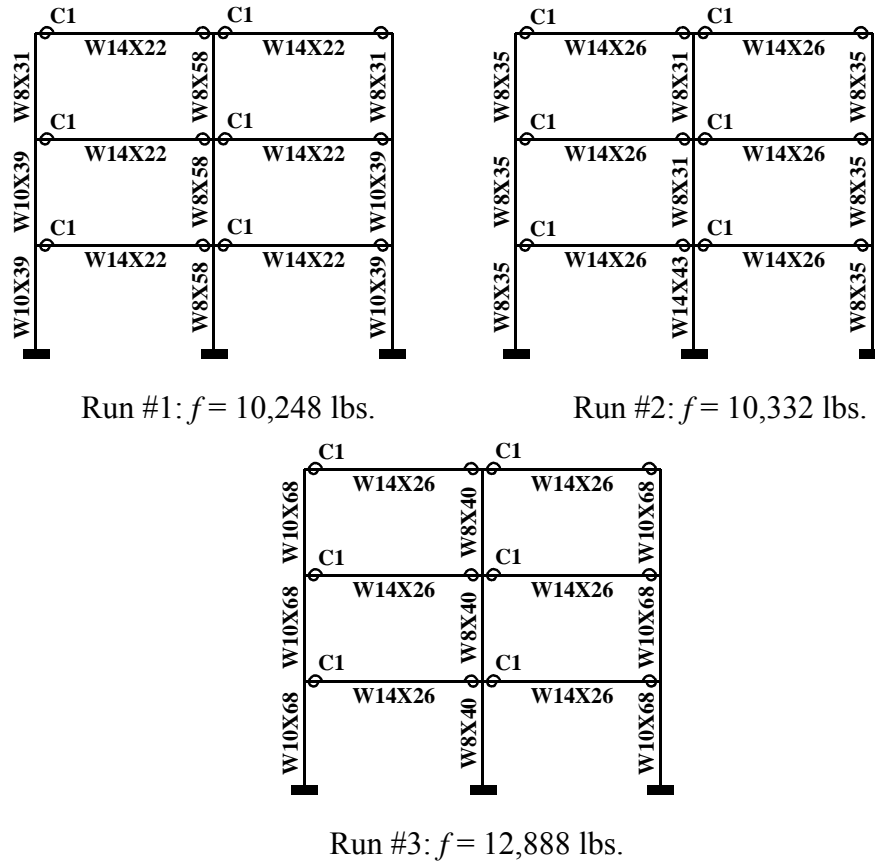


Figure 5.3: Evolved Configurations for Frame 1 with FR Connections

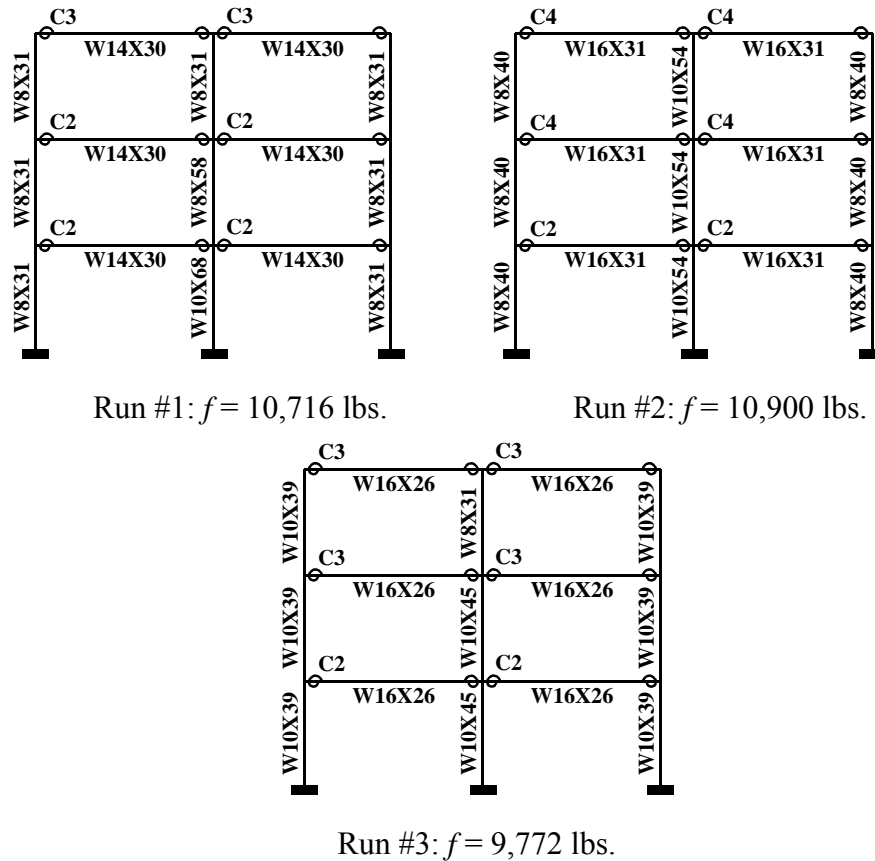


Figure 5.4: Evolved Configurations for Frame 1 with PR Connections

According to Figure 5.5, the proposed algorithm over estimates the ultimate strength of the frame, however, it is conservative with respect to the displacement demands imposed on the members and connections. The lateral displacement corresponding to the less stringent parameters ($N_{fe,col} = 10$; $N_{fe,bm} = 10$) exceeds that in the more stringent analysis ($N_{fe,col} = 20$; $N_{fe,bm} = 30$). The plastic hinge rotations and connection rotations are expected to be greater in the less accurate ultimate load prediction. More importantly, recall that the constraint criteria are evaluated for the attained load factor (service and ultimate) and if a frame does not attain the prescribed factored load combination, it is penalized accordingly. Therefore, the accuracy of the

algorithm really only needs to be assessed for load levels less than or equal to a load factor of one. Based on the representative load response in Figure 5.5, the proposed algorithm possesses sufficient accuracy.

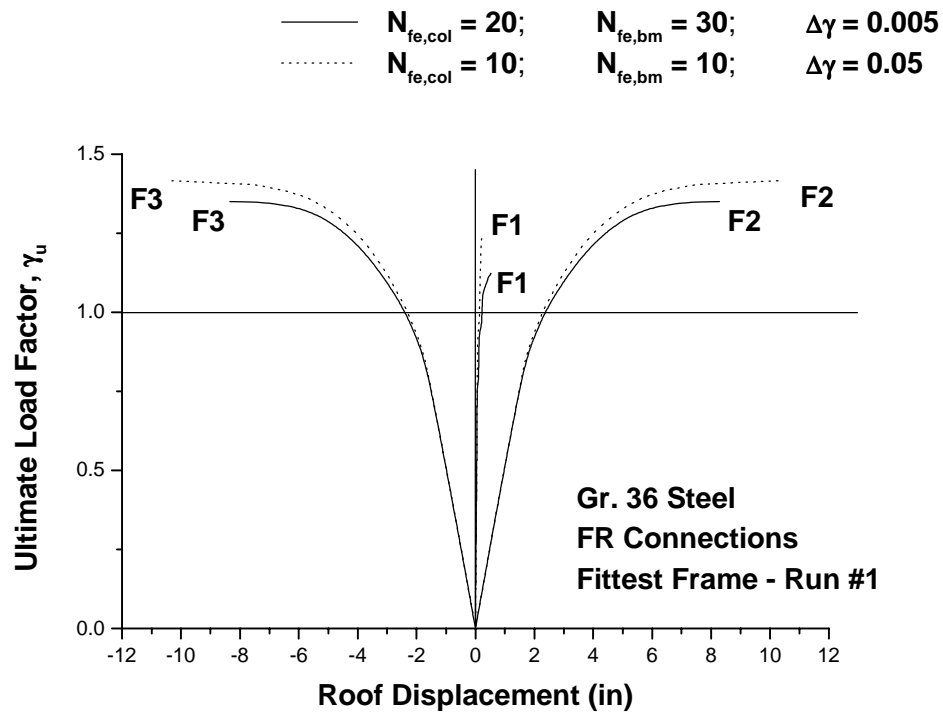


Figure 5.5: Assessment of Evaluation Parameters for Frame 1

The evolved designs were compared with previous research in order to measure the capability of the proposed algorithm to achieve optimized designs. Table 5.2 compares the member weights (excluding connection weight factors) for the lightest FR (Run #1) and PR (Run #3) frames with the results from Kameshki and Saka (1999). Connection weight factors were not included in the comparison due to the discrepancy between the PR (semi-rigid) connection models. (Kameshki and Saka 1999) used a polynomial model to represent all PR (semi-rigid) connections as end plates without

column stiffeners. Based on the discussion provided by Kameshki and Saka (1999), the connection behavior appears to be qualitatively similar to the C3 and C4 found in the evolved frames.

Table 5.2: Weight Comparison for Frame 1

Frame	Kameshki et. al. (1999)	Proposed Design Algorithm
FR	9,744 lbs.	7,344 lbs.
PR	8,640 lbs.	7,380 lbs.

A comparison of the structural behavior of the lightest FR (Run #1) frame and PR (Run #3) frame was conducted using the ultimate load response versus the characteristic roof displacement plots in Figure 5.6. It is interesting to note the slight nonlinear response of the frames as they approach the target applied load ratios for the lateral load cases, which would not be captured with a conventional linear-elastic analysis. In addition, the PR frame achieves a higher ultimate load ratio than the FR frame. The fittest FR frame (Figure 5.3, Run #1) and PR frame (Figure 5.4, Run #3) are similar in weight and member sizes with the slightly larger beams of the PR frame balancing the less stiff connections.

Penalty convergence plots were used to determine which constraint criteria the proposed algorithm worked on during the frame's evolution. The number of individuals violating each constraint were tallied for each generation. The constraint criteria were divided into serviceability constraints (Figure 5.7) and strength constraints (Figure 5.8). It is evident from the plots that the proposed algorithm had difficulty satisfying the horizontal deflection (δ_H) at service load levels. Lateral-torsional buckling unbraced

length (L_{pd}) and shape criteria (shape) were being worked on at ultimate load levels. This behavior implies that the lateral load cases (F2 and F3) heavily influence the design.

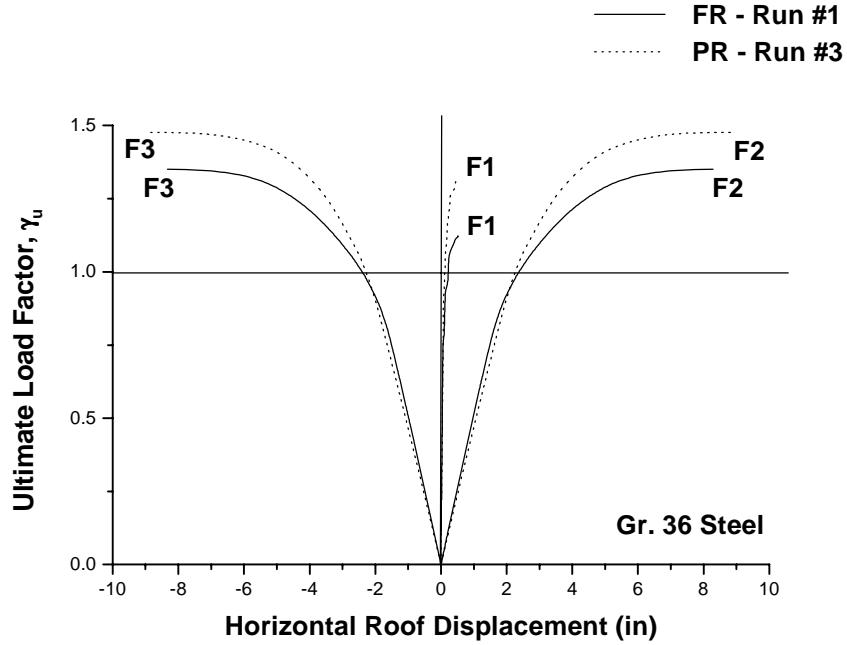


Figure 5.6: Load Deformation Response for Frame 1

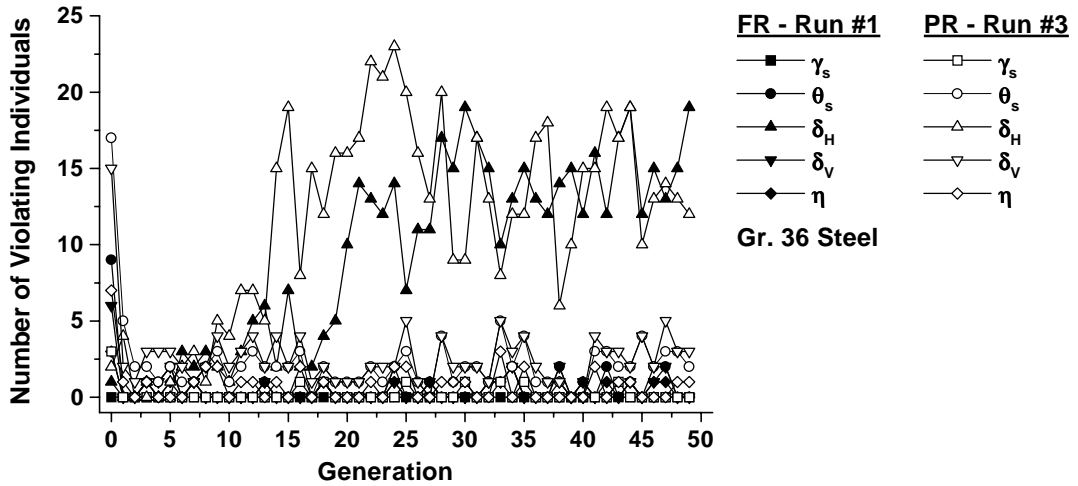


Figure 5.7: Service Penalty Convergence for Frame 1

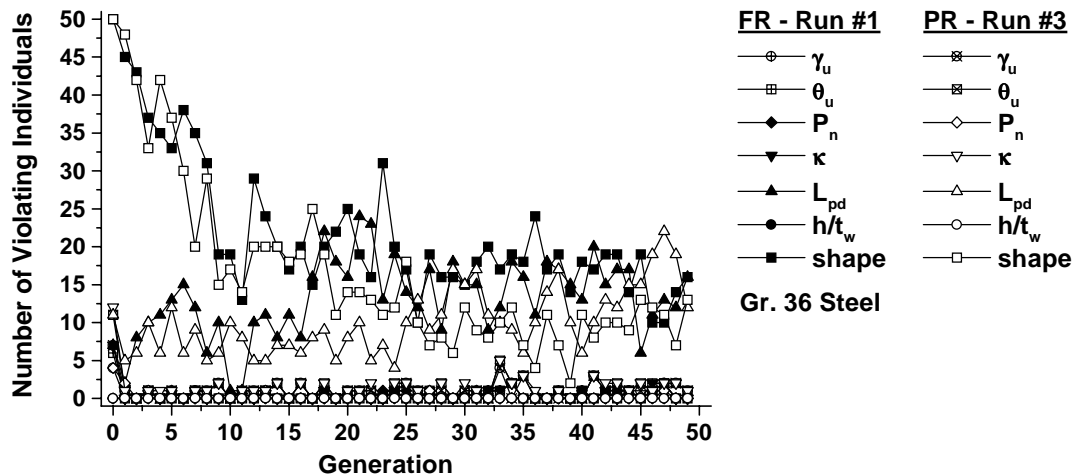


Figure 5.8: Ultimate Penalty Convergence for Frame 1

5.2 Frame 2 – Two-Story, Three-Bay

The second frame utilized to evaluate the proposed algorithm is the two-story, three-bay frame shown in Figure 5.9. The frames are spaced at 20 feet on center with a wind pressure of 20 psf. The steel is assumed to have a modulus of elasticity of 29,000 ksi. Additional design parameters are provided in Table 5.3.

Both FR and PR connections were investigated along with an assessment of the effect of steel strength on the design. Five variations of the frame were designed: (a) FR connections with Grade 36 steel; (b) PR connections with Grade 36 steel; (c) FR connections with Grade 50 steel; (d) PR connections with Grade 50 steel; and (e) FR connections with Grade 50 steel and a 12 foot first story height. Three design runs were conducted for each variation. Comparisons of the structural behavior with respect to the connection types and steel grades will be made using load response curves and penalty

convergence plots. In addition, the influence of the 1st story height on the design will be analyzed for frames with Grade 50 steel and FR connections.

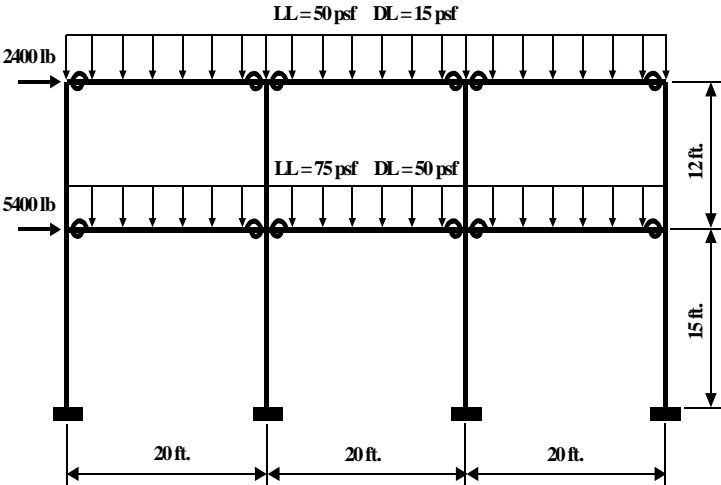


Figure 5.9: Topology and Loading for Frame 2

Table 5.3: Design Parameters for Frame 2

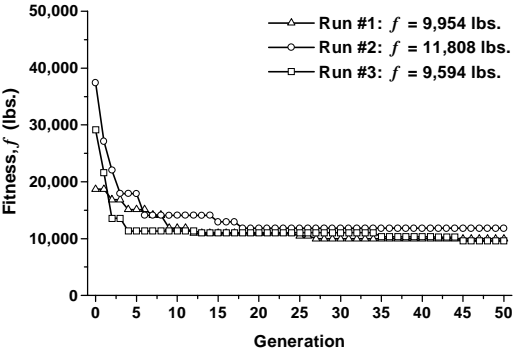
Evolutionary Parameters:			Evaluation Parameters:	
Grouped Design Variables	6 (FR), 8 (PR)		Number of Sub-columns	10
Population Size	50		Number of Sub-beams	10
Number of Generations	50		Load Increment	0.05
Reproduction Parameters:			Selection Parameters:	
	<u>Crossover</u>	<u>Mutation</u>	Tournament Size	2
Building Rate	na	*	Partition Break	0.4
Story Rate	na	*	Probability of Higher Fitness	0.7
Column Rate	0.6	0.3	Elitism	on
Beam Rate	0.6	0.3	Penalty Function	linear
Connection Rate**	0.6	0.3	Penalty Rate	5

* Suppressed (*i.e.* Rate = 0.001)
 ** Suppressed for FR connection runs

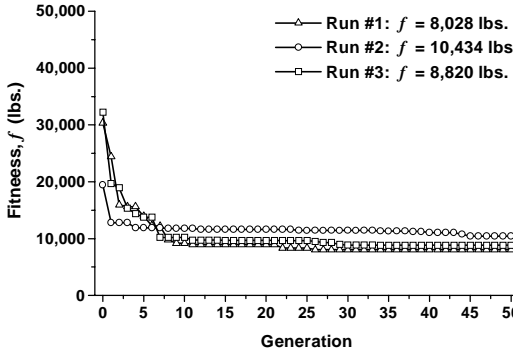
The design parameters were consistent for each design run with the exception of the mutation parameters for the frames with FR connections and Grade 50 steel. Two of the design runs for the frame with a 15-foot first story height, variation (c) used a story mutation rate of 30% and each of the three design runs for the frame with a 12-foot story height (variation e), used a story mutation rate of 10%. Contrary to the limited parameter study in Chapter 2, the disparity in story mutation rates did not significantly influence the evolutionary process as illustrated with the relatively early stagnation of the fitness trajectories shown in Figure 5.10.

Similar to Frame 1, the fitness trajectories (*i.e.* modified weight adjusted to account for connections and constraint violations) start with heavier frames and converge to a lighter frame. However, the initial fitness values of the frames in first generation are significantly less in Frame 2 (approximately 40,000 lbs.) compared to Frame 1 (approximately 200,000 lbs), which can be attributed to the higher probability of violating the constructability criteria (shape constraint) due to the three-story stacking of columns in Frame 1. Also, the random nature of the proposed algorithm is evident with the large range of initial frame fitness for Frame 1 with Grade 50, FR connections, and a 12-foot 1st story height (variation e). The heaviest initial frame was 58,902 lbs. and the lightest frame was 11,352 lbs.

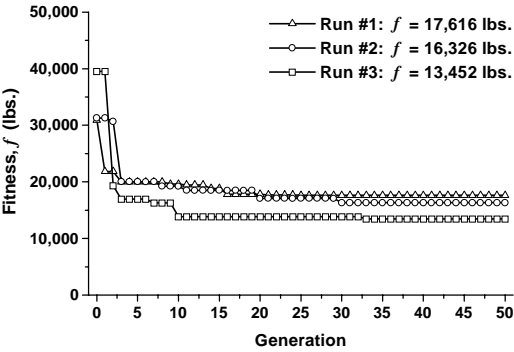
Repeatability of the results was not attained as seen in the evolved configurations in Figures 5.11 through 5.15. As stated in discussion of Frame 1 results, these designs can be viewed as “optimized” demonstrating the possibility for multiple local minimums within the solution space of possible feasible designs.



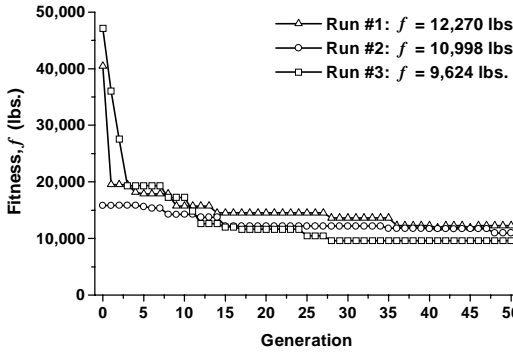
(a) Grade-36 Steel with FR Connections



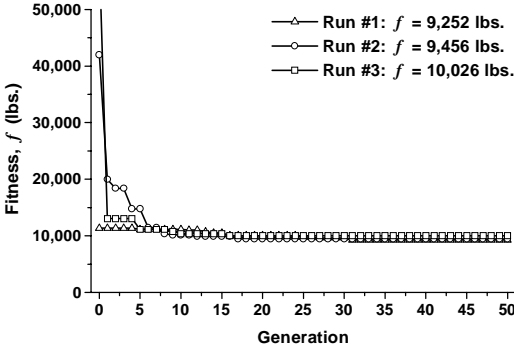
(b) Grade-36 Steel with PR connections



(c) Grade-50 Steel with FR Connections



(d) Grade-50 Steel with PR Connections



(e) Grade 50 Steel with FR Connections and 12 foot 1st Story Height

Figure 5.10: Fitness Trajectories for Frame 2

The evolved configurations result in reasonable member sizes with the exception of the FR frames with Grade 50 steel in Figure 5.13, which consistently resulted in significantly larger columns and smaller beams when compared to the PR frames with Grade 50 steel in Figure 5.14. This discrepancy in results was the motivation for reducing the 1st story height to 12 feet for the FR frames with Grade 50 steel in Figure 5.15, which resulted in more reasonable member sizes. Further discussion of the discrepancies in the results for frames with Grade 50 steel will be provided in subsequent paragraphs.

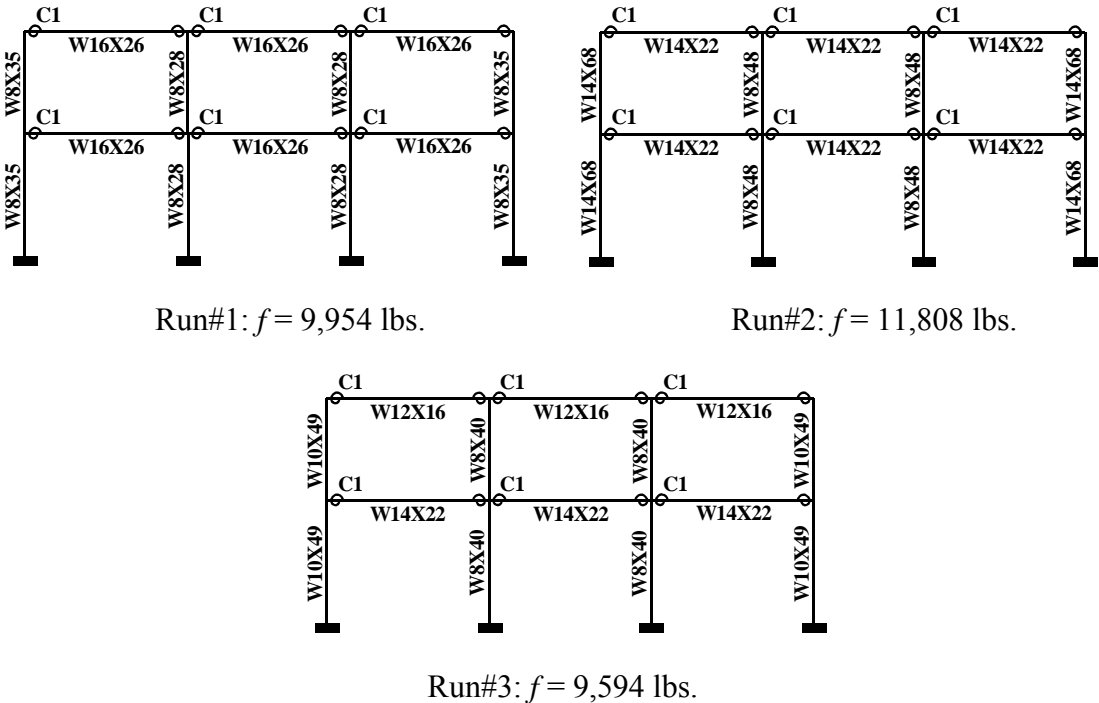
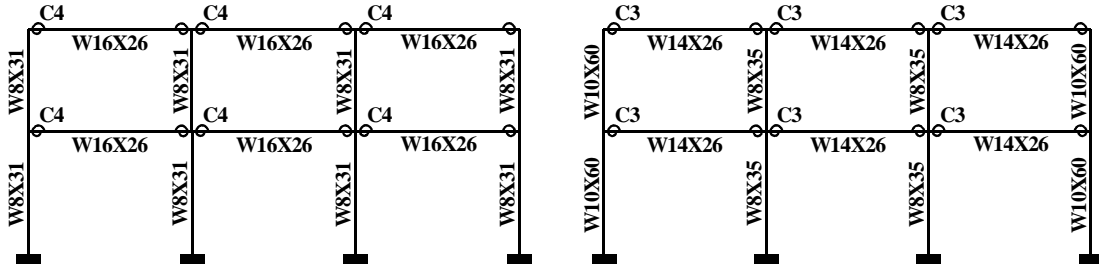
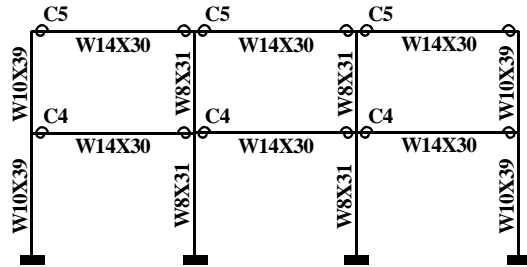


Figure 5.11: Evolved Configurations for Frame 2 with Grade 36 Steel and FR Connections



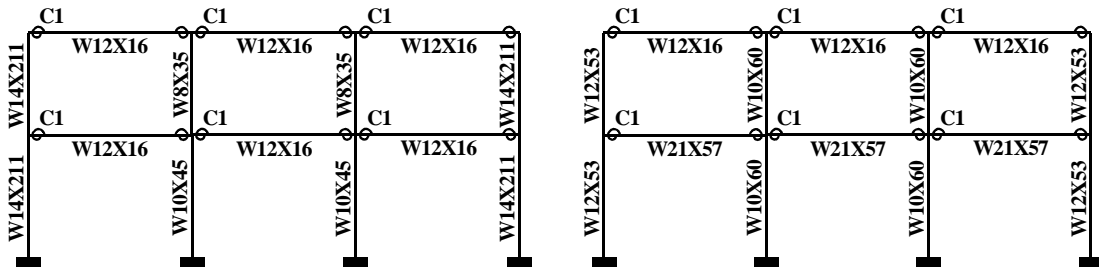
Run #1: $f = 8,028$ lbs.

Run #2: $f = 10,434$ lbs.



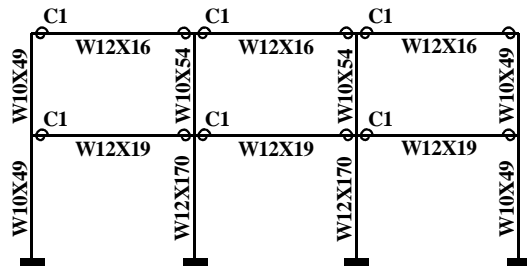
Run #3: $f = 8,820$ lbs.

Figure 5.12: Evolved Configurations for Frame 2 with Grade 36 Steel and PR Connections



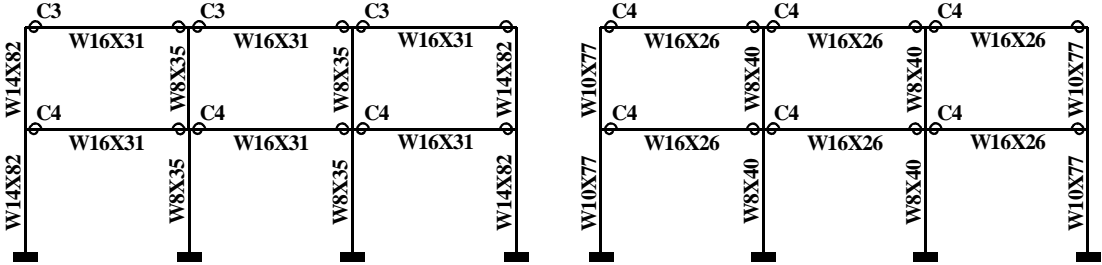
Run #1: $f = 17,616$ lbs.

Run #2: $f = 16,326$ lbs.



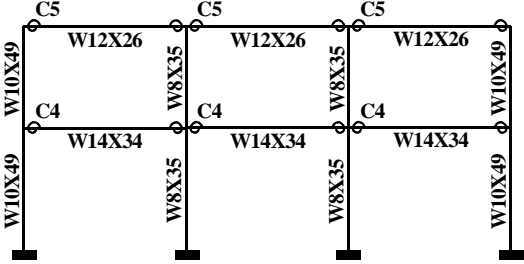
Run #3: $f = 13,452$ lbs.

Figure 5.13: Evolved Configurations for Frame 2 with Grade 50 Steel and FR Connections



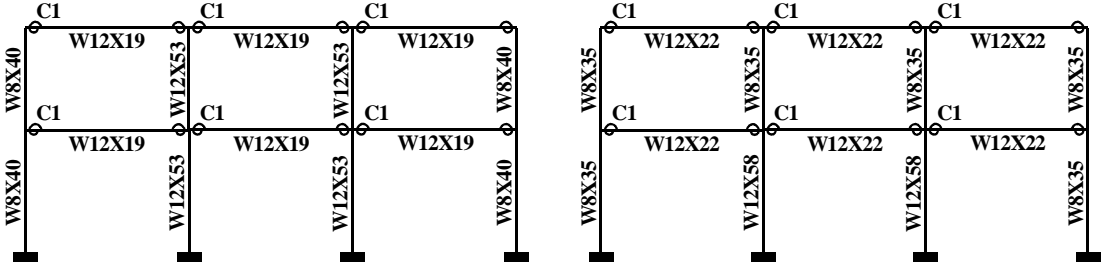
Run #1: $f=12,270$ lbs.

Run #2: $f= 10,998$ lbs.



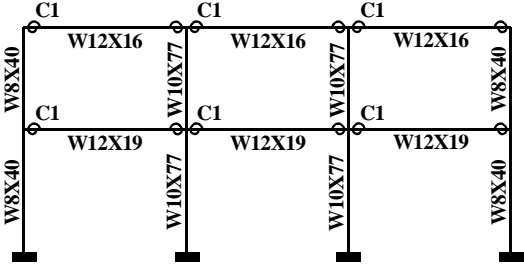
Run #3: $f= 9,624$ lbs.

Figure 5.14: Evolved Configurations for Frame 2 with Grade 50 Steel and PR Connections



Run #1: $f= 9,252$ lbs.

Run #2: $f= 9,456$ lbs.



Run #3: $f=10,026$ lbs.

Figure 5.15: Evolved Configurations for Frame 2 with Grade 50 Steel, FR Connections, and 12 ft. 1st Story Height

As stated in the discussion of Frame 1, the accuracy and duration of the evolutionary process is influenced by the design parameters outlined in Table 5.3. Similar to Frame 1, run times for each evolutionary design were approximately six hours (wall clock) using a Pentium-based personal computer with a 400 MHz processor speed. Figure 5.16 illustrates the impact of more stringent parameters on the ultimate load versus characteristic roof displacement. The proposed algorithm overestimates the ultimate strength of the frame, but also conservatively overestimates the demand on the members and connections within the frame similar to the results for Frame 1. More importantly, the response is essentially the same for load levels less than or equal to the target ultimate load cases ($\gamma_u = 1$).

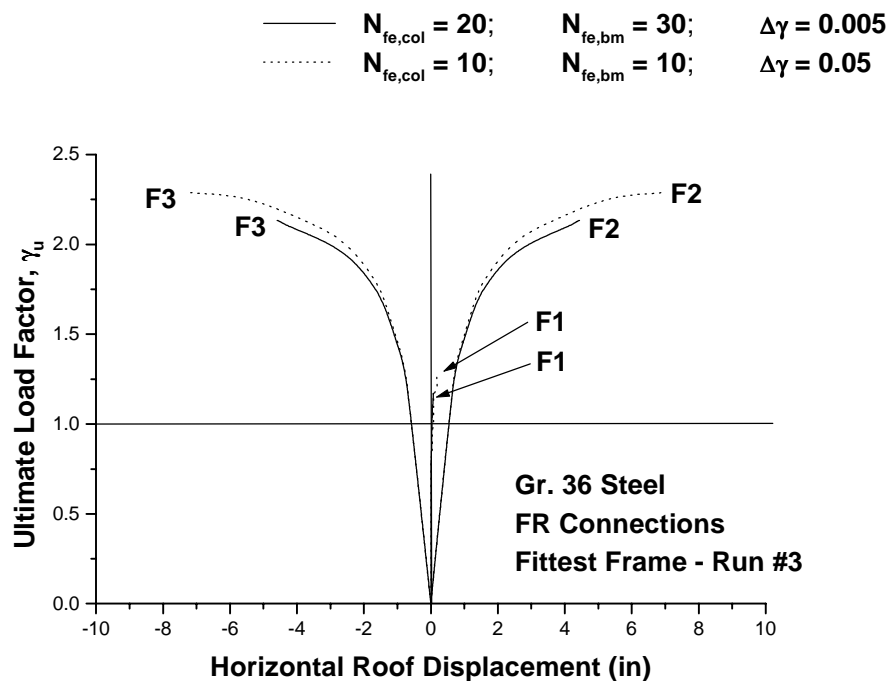


Figure 5.16: Assessment of Evaluation Parameters for Frame 2

The weight of the members without consideration of the connections for the fittest FR (Run #3) and PR (Run #1) frames with Grade 36 steel were compared to the results obtained by Xu et al. (1995) in Table 5.4. In order to justify the weight comparisons for the PR frames, the non-dimensionalized connection strengths were calculated for the extended end plate connections found in Xu et al. (1995). The design variables were grouped in the same manner as the proposed algorithm, thus the beams and connections were the same at each story. Using the connection stiffness and beam sizes provided (Xu et al. 1995), the non-dimensionalized moment capacities of the 1st and 2nd story connections were calculated to be 0.65 and 0.48, respectively. These results envelope the 0.55 non-dimensionalized moment capacity of the C4 connection found in the fittest evolved configuration shown in Figure 5.12, Run #1.

Table 5.4: Weight Comparison for Frame 2

Frame	Xu, et. al. (1995)	Proposed Design Algorithm
FR	7,031 lbs.	7086 lbs.
PR	6,712 lbs.	6468 lbs.

The effect of the grade of steel on the design of the frames is demonstrated in the load deformation response curves in Figure 5.17 and Figure 5.18. One should note the seemingly linear response of the frames for load levels up to the ultimate load conditions ($\gamma_u = 1$) in the case of Grade 50 steel with FR connections and all frames under gravity loads (F1). In addition, the frames achieve a significantly smaller ultimate load level under the gravity load case (F1) with respect to the lateral load cases (F2 and F3) implying gravity-controlled designs. Grade 50 steel results in a stronger frame compared

to Grade 36 steel for the frames with PR connections as shown in Figure 5.18. However, the strengths of frames with FR connections are similar for the both steel grades as shown in Figure 5.17. Upon review of the evolved Grade 50 frame with FR connections (Figure 5.13 Run #3) it is apparent a beam failure mechanism occurred, preventing nonlinear lateral sway behavior from developing as in the frame with Grade 36. Thus, the beam mechanism formation near the ultimate load case ($\gamma_u = 1$) yields a load deformation response similar to the gravity load (F1) case.

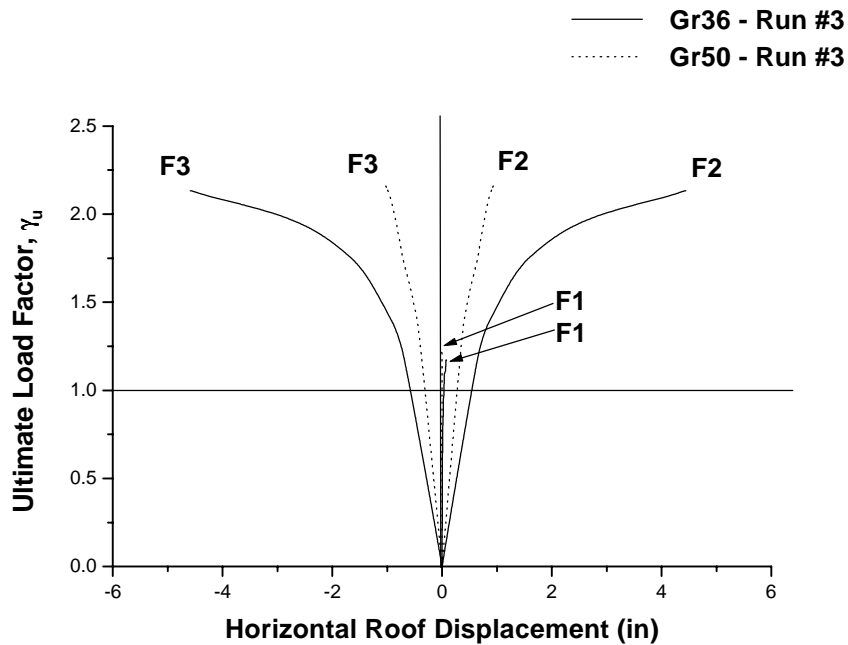


Figure 5.17: Load Deformation Response for Frame 2 with FR Connections

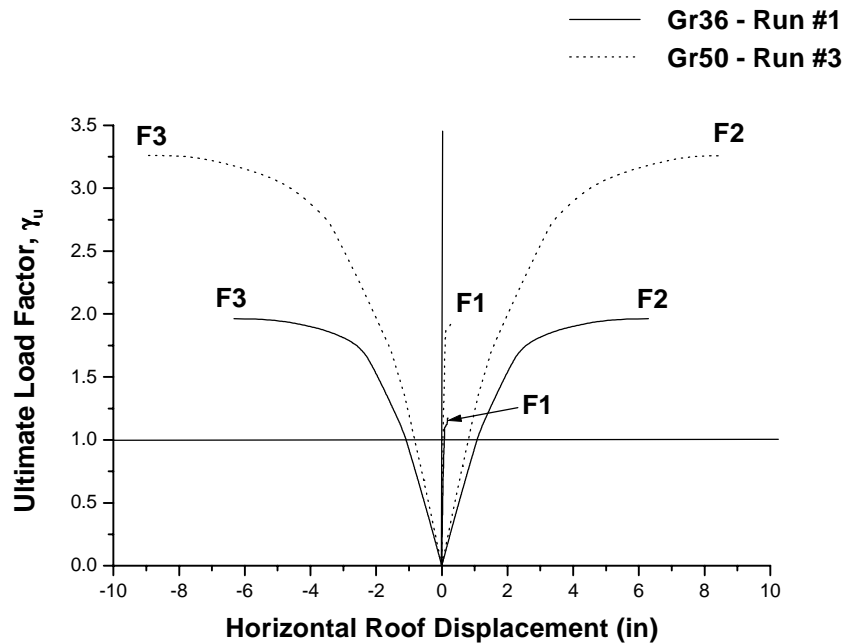


Figure 5.18: Load Deformation Response for Frame 2 with PR Connections

The influence of connection restraint on the ultimate load response of frames with Grade 36 steel is shown in Figure 5.19. In contrast to the Frame 1 results, the frame with FR connections achieves a higher ultimate load ratio than the frame with PR connections. Also, the response of the frames remains linear as they approach the ultimate applied load ratios for the lateral load cases. The fittest PR frame (Figure 5.12, Run #1) is significantly lighter than the fittest FR frame (Figure 5.11, Run #3). Although, the PR frame requires larger beams to balance the flexibility of the connections, it does not require the larger columns of the FR frame necessary to transfer the larger moment attracted by the connections.

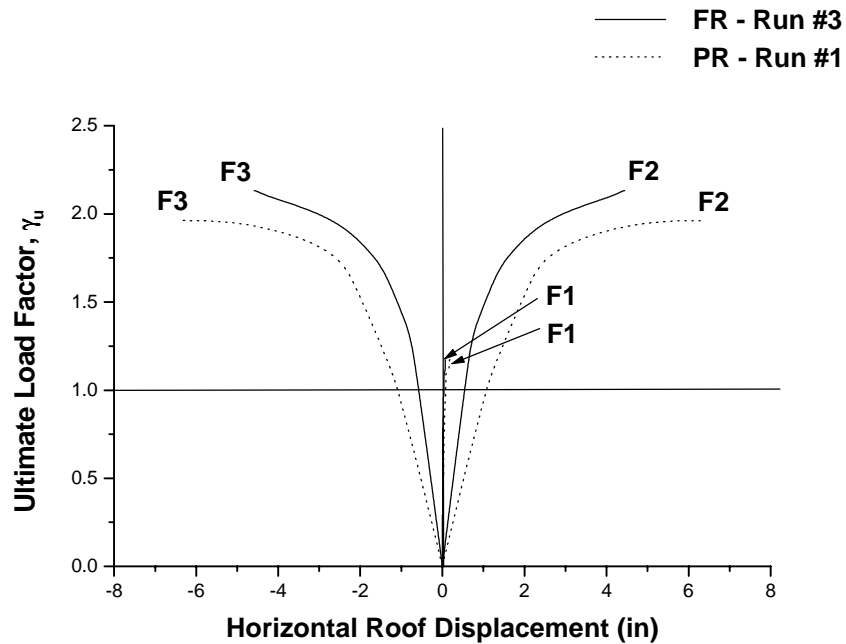


Figure 5.19: Load Deformation Response for Frame 2 with Grade 36 Steel

The participation of the constraints throughout the evolution was again considered using penalty convergence trajectories. Figure 5.20 through 5.23 present the number of violating individuals for each service and ultimate load constraint at each generation of the evolutionary process. The beam deflection (δ_v) and connection rotations (θ_s) dominate the serviceability requirements. However, the lateral-torsional buckling (L_{pd}) and constructability (shape) constraints are controlling the overall design for both Grade 36 and Grade 50 steel. Notice that these ultimate load and constructability constraints have a greater impact on the Grade 50 steel (Figure 5.23) than the Grade 36 steel (Figure 5.21).

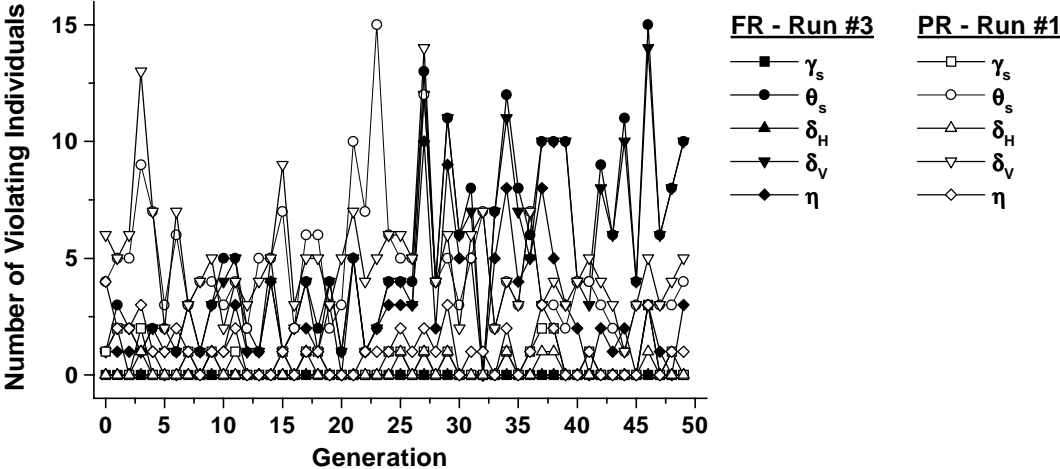


Figure 5.20: Service Penalty Convergence for Frame 2 with Grade 36 Steel

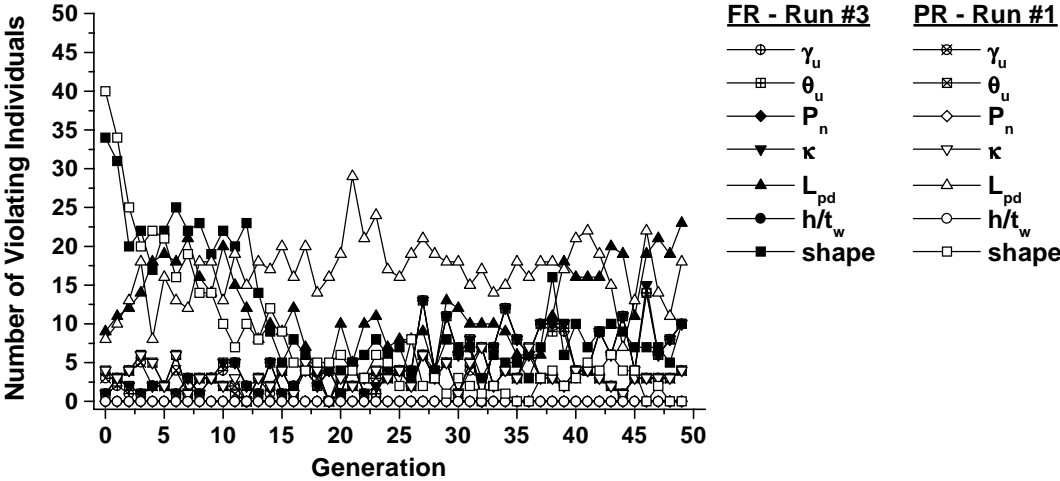


Figure 5.21: Ultimate Penalty Convergence for Frame 2 with Grade 36 Steel

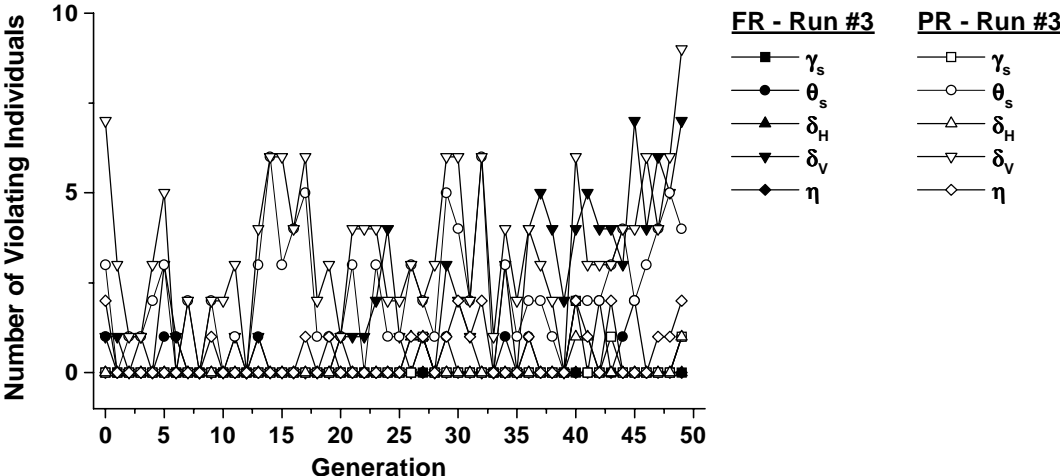


Figure 5.22: Service Penalty Convergence for Frame 2 with Grade 50 Steel

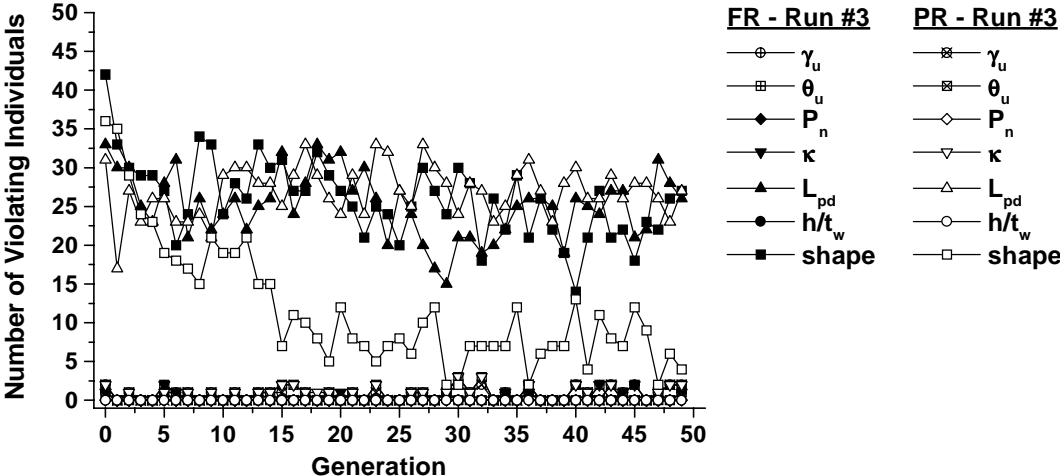


Figure 5.23: Ultimate Penalty Convergence for Frame 2 with Grade 50 Steel

An investigation into the influence of the 1st story height for Frame 2 with Grade 50 steel and FR connections was undertaken. Upon changing the 1st story height from 15 feet to 12 feet, more reasonable member sizes were obtained. Furthermore, Figure 5.24 illustrates the difference in the characteristic horizontal roof displacement response under

ultimate loading for the different story heights. The jaggedness of the response does not represent pure lateral sway instability. This can be attributed to the interaction between the gravity loads deflecting the column top at the roof in the opposite direction to the lateral load's imposed deformation at the column top. The frame with a 12-foot 1st story height achieves a higher lateral load ratio than the 15 foot 1st story height. Also, the frame with a 12 foot story height (Figure 5.15, Run #1) is more efficient (*i.e.* lighter) and practically sized compared to the frame with a 15 foot story height (Figure 5.13, Run #3).

Upon review of the comparison of ultimate penalty trajectories in Figure 5.26, it is apparent the proposed algorithm does not struggle as much with the lateral-torsional buckling (L_{pd}) and constructability (shape) constraints for a 1st story height of 12 feet when compared to the 15-foot 1st story height. In addition, the beam deflection (δ_v) and connections (θ_s) remain the most influential serviceability criteria.

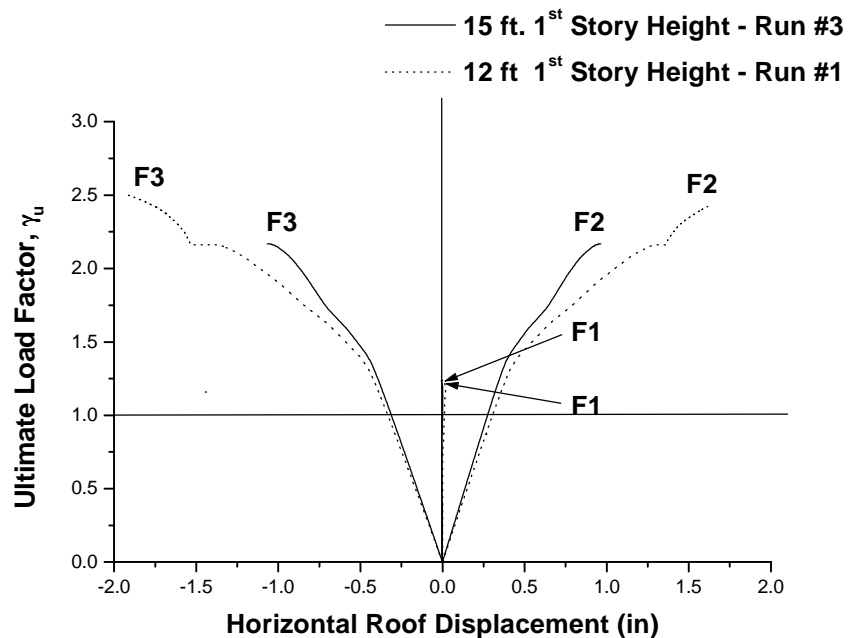


Figure 5.24: Load Deformation Response for Frame 2 with Grade 50 Steel and FR Connections

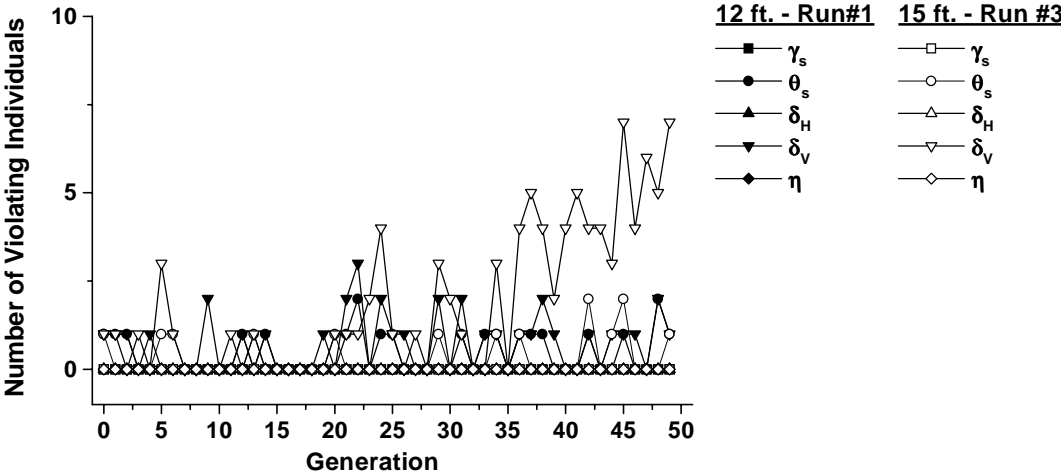


Figure 5.25: Service Penalty Convergence for Frame 2 with Grade 50 Steel

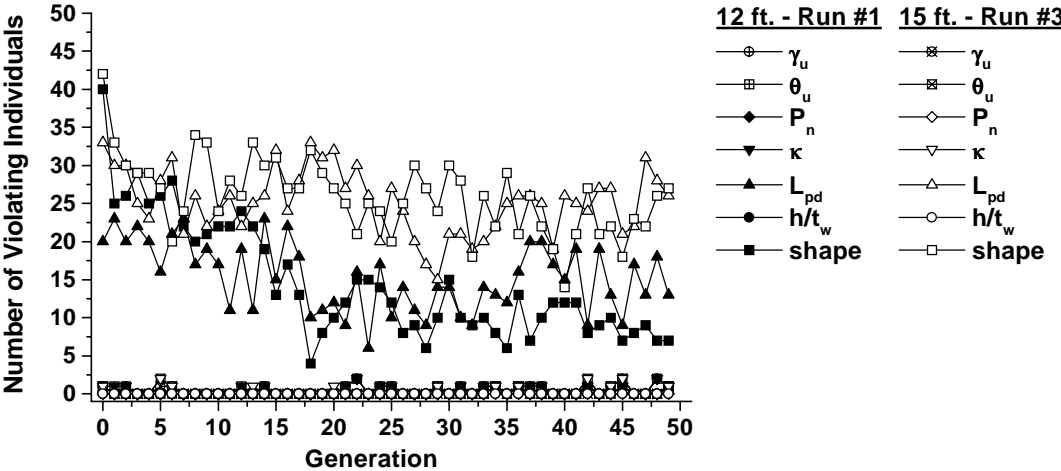


Figure 5.26: Ultimate Penalty Convergence for Frame 2 with Grade 50 Steel

5.3 Frame 3 - Ten-Story, Three-Bay

The objective of this section is to measure the applicability of the proposed algorithm to larger frames using the ten-story, three-bay frame studied by Xu and Grierson (1993).

The frame topology and loading is shown in Figure 5.27. The frames are spaced every 30

feet with a wind pressure of 15 psf with an additional 7.8 psf applied to the top story (*i.e.* half applied to the roof and half applied to the 9th story). The modulus of elasticity and yield strength of the steel assumed to be 29,000 ksi and 36 ksi, respectively. Additional design parameters are presented in Table 5.5.

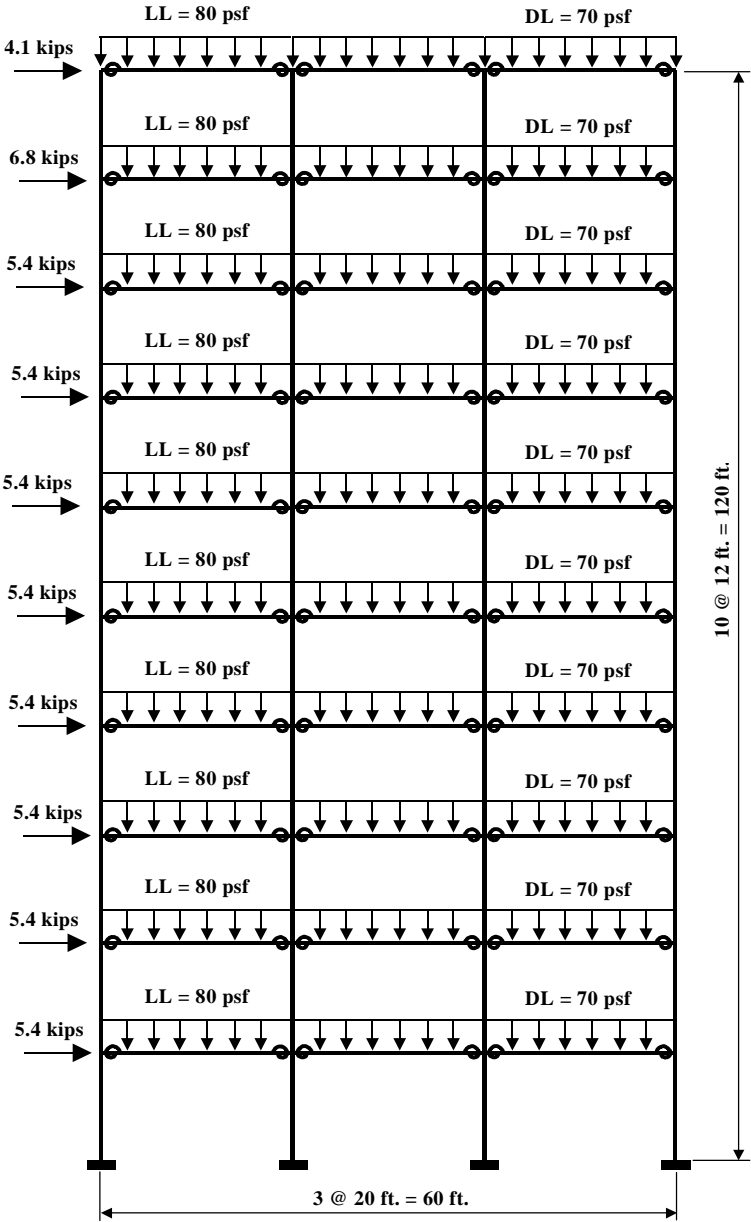


Figure 5.27 Topology and Loading for Frame 3

Table 5.5: Design Parameters for Frame 3

Evolutionary Parameters:			Evaluation Parameters:	
Grouped Design Variables	30 (FR), 40 (PR)		Number of Sub-columns	10
Population Size	50		Number of Sub-beams	10
Number of Generations	50		Load Increment	0.05
Reproduction Parameters:			Selection Parameters:	
	Crossover	Mutation	Tournament Size	2
Building Rate	na	0.05	Partition Break	0.4
Story Rate	na	0.05	Probability of Higher Fitness	0.7
Column Rate	0.75	0.05	Elitism	on
Beam Rate	0.75	0.05	Penalty Function	linear
Connection Rate*	0.75	0.05	Penalty Rate	2

* Suppressed for FR connection runs

The frame was design for two conditions: (a) FR connections and (b) PR connections. Three designs were run for each of the conditions. The performance of the proposed algorithm and structural behavior of the fittest frames for both connection types will be investigated via fitness trajectories, load response curves, and penalty convergence plots.

The fitness trajectories in Figure 5.28 illustrate similar stable convergence found in the previous studies, however, the stagnant points occur at a later stage in the evolution (between 20 to 25 generations).

The proposed algorithm did not achieve repeatability in the results as seen in the evolved configurations presented in tabular form in Figures 5.29 and 5.30. The prevailing characteristic of each frame is the consistent column size for the entire height of the building. The motivation for the constructability constraint was to encourage the proposed algorithm to evolve frames with telescoping column configurations (decreasing columns size and weight with increasing story heights). Apparently the proposed

algorithm located a preferred column shape and propagated it up the entire height of the building via non-homologous crossover. Another observation is the variation in beam sizes over the height of the frame, which is more evident in the PR frames. This phenomenon has been encountered by previous researchers (Camp et al. 1996, Kameshki and Saka 1999, Xu and Grierson 1993). The algorithms struggle to strike a balance between the connections, beams, and the adjacent columns the case of PR connections.

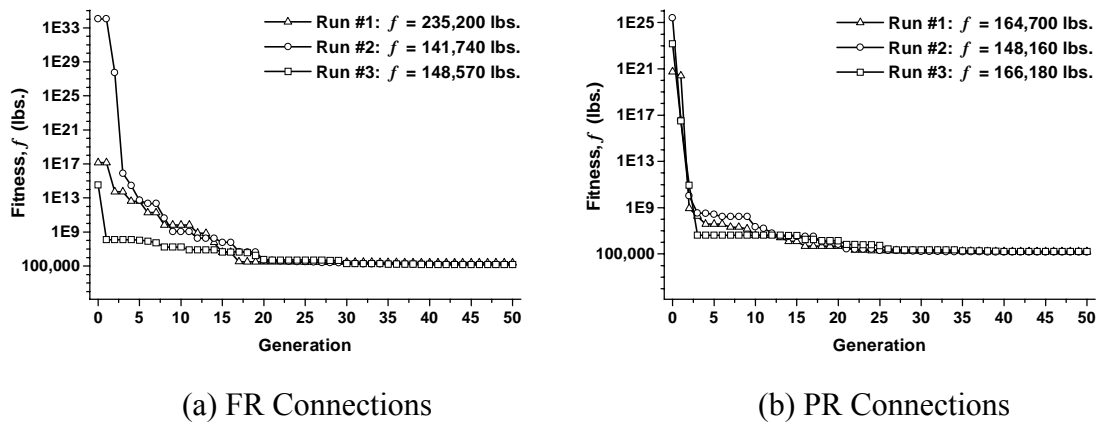


Figure 5.28: Fitness Trajectories for Frame 3

Figure 5.31 illustrates their impact of the evaluation parameters outlined in Table 5.5 on the inelastic response. The results are similar to those obtained in the previous studies, in which the response is essentially identical for load levels less than or equal to the target ultimate load condition ($\gamma_u = 1$). An interesting observation is the large roof displacement under the gravity load condition (F1) due to the applied notional loads needed to account for member imperfections, which is amplified with the taller building.

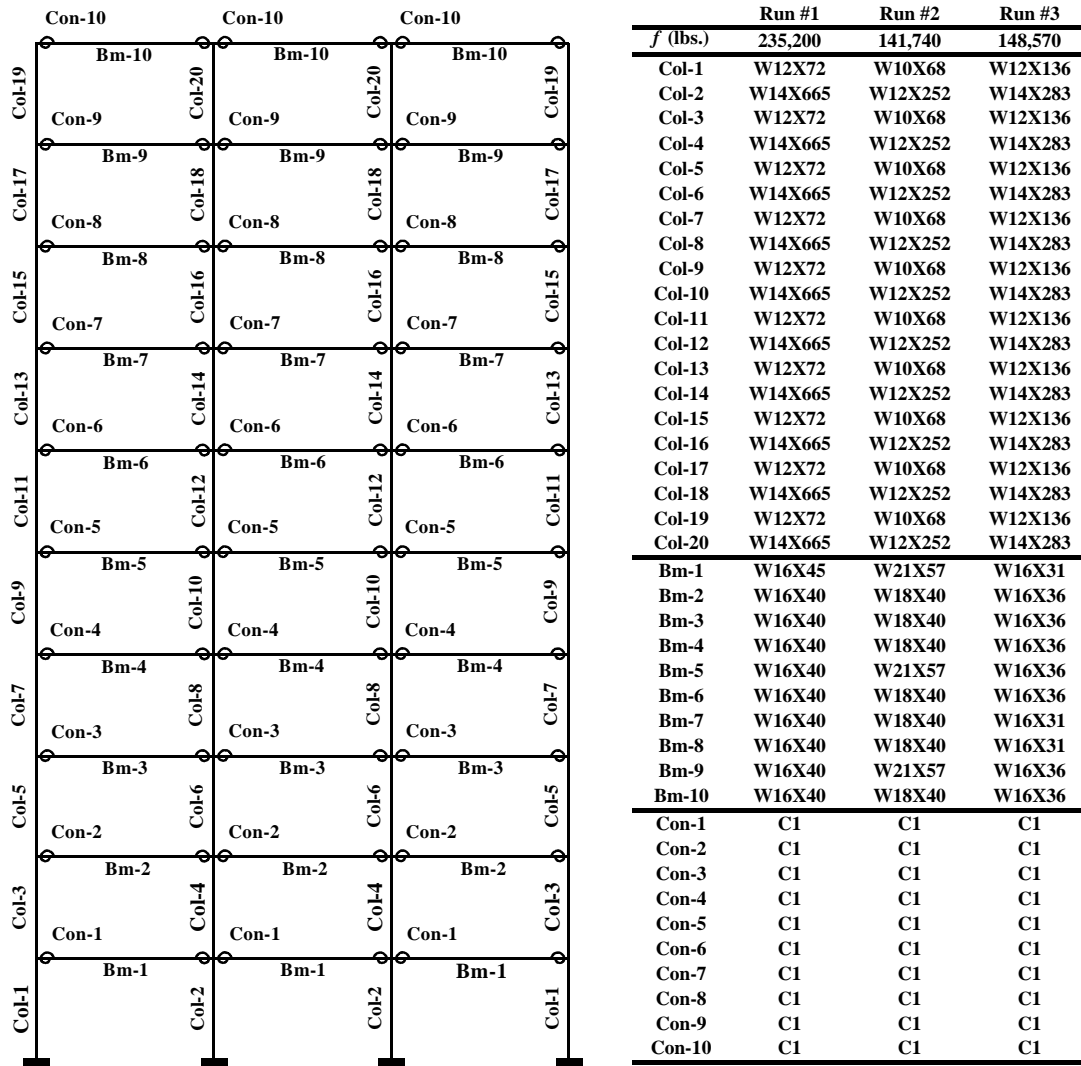


Figure 5.29: Evolved Configurations for Frame 3 with FR Connections

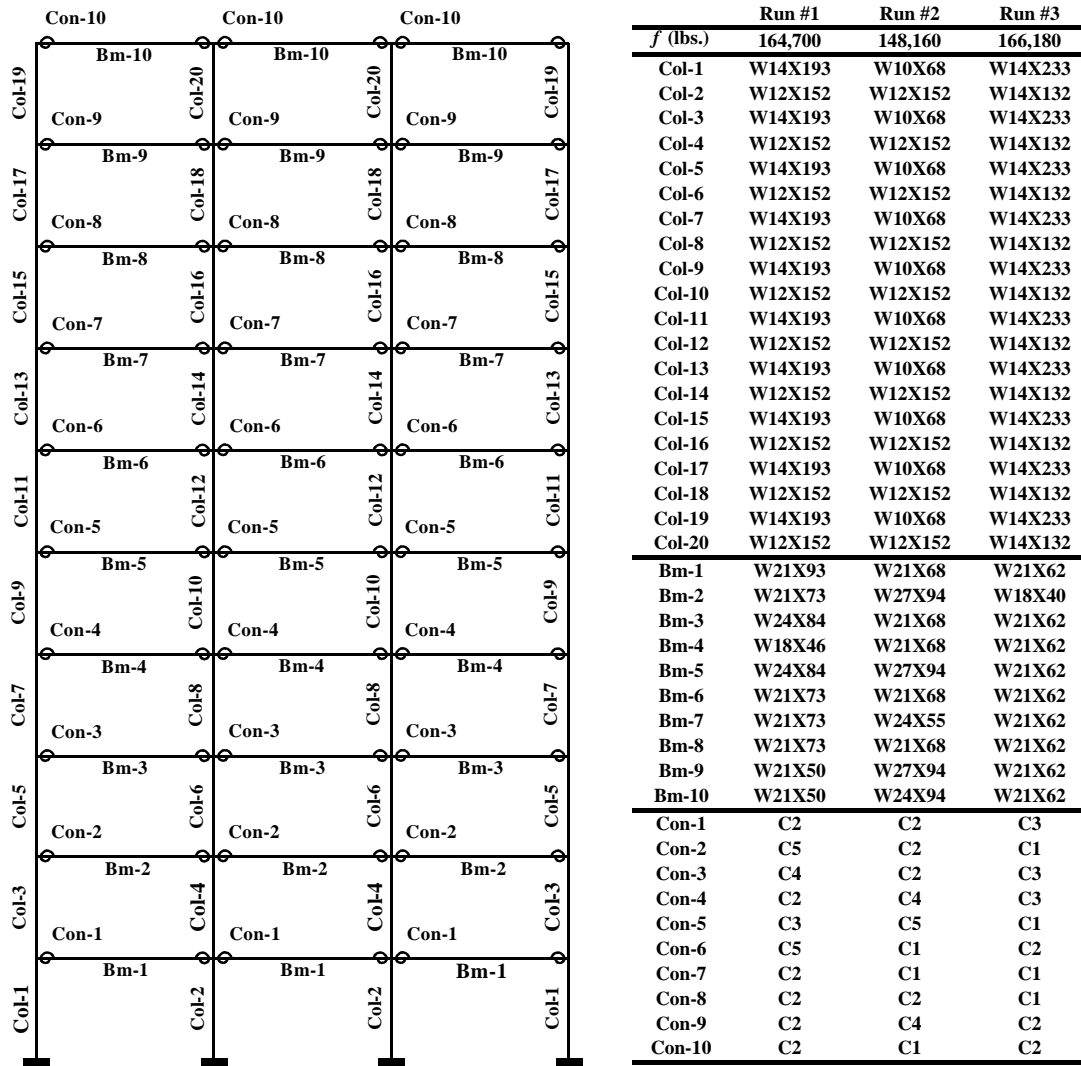


Figure 5.30: Evolved Configurations for Frame 3 with PR Connections

The evaluation parameters and size of the frame also significantly impact the duration of the evolutionary process. The runtime of the inelastic analysis increases as the number of finite elements increases and the initial load increment decreases. In addition, the duration of the evolutionary process increases with the size of the frame. The evolutionary algorithm for Frame 3 took approximately 8 hours and 24 hours for processor speeds of 1400 MHz and 400 MHz, respectively.

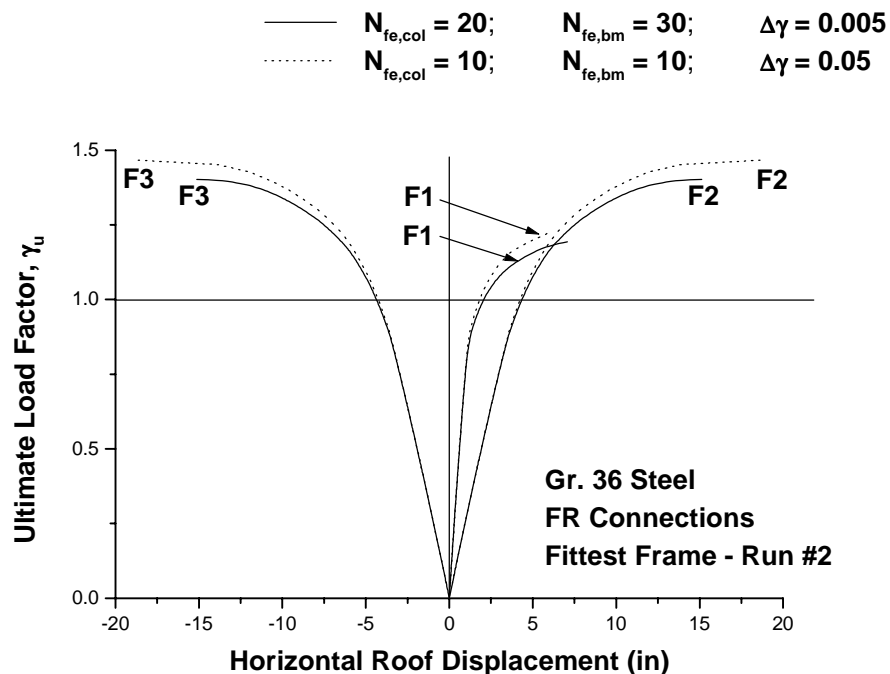


Figure 5.31: Assessment of Evaluation Parameters for Frame 3

The adequacy of the evolved design for Frame 3 was gauged by comparing the member weights with the results obtained by Xu and Grierson (1993). Table 5.6 presents a breakdown of the total weight of the beams, total weight of the columns, and the total weight of the frame for the fittest PR frame (Run #2) and the partially restrained frame

designed by Xu and Grierson (1993). Fully restrained frame results were not provided in Xu and Grierson (1993), thus no comparison was made.

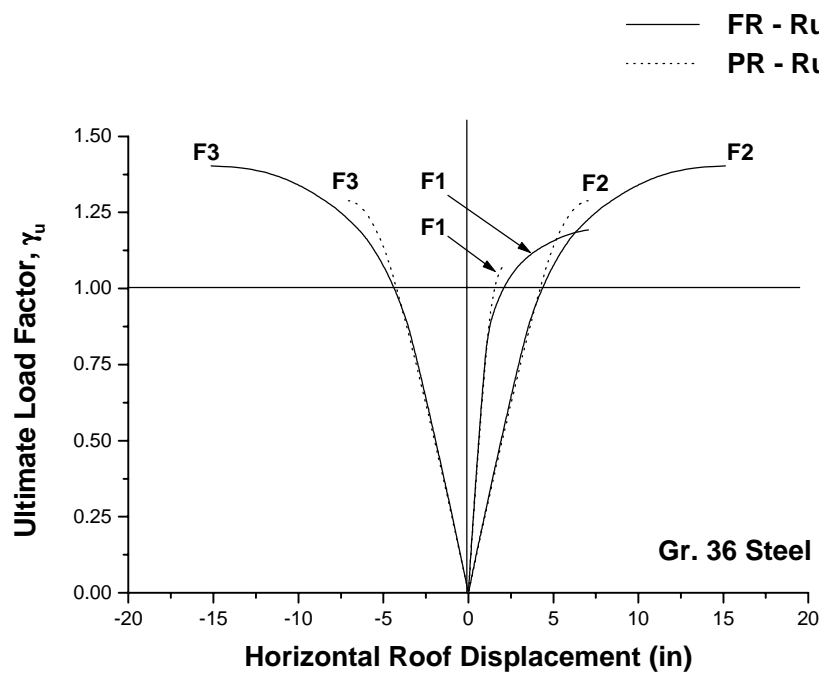
A comparison of the connection stiffness for each frame was conducted to validate the result comparison. Since the design variables were not grouped in the same manner, the connection stiffness at each story could not be compared directly. Instead, a comparison of the total connection stiffness for each frame was conducted by summing up the stiffness of each connection within each frame. Xu and Grierson (1993) provided a secant stiffness of each connection, therefore, an equivalent secant stiffness for each connection type found in the fittest PR frame (Run #2) was needed. The equivalent stiffness for each connection type in the optimized PR frame was defined as the slope of the line extending from the origin to the second kink of the moment-rotation curves in Figure 3.2. Each connection was non-dimensionalized with respect to the connected beam and its stiffness was summed up to arrive at the total connection stiffness for the fittest PR frame (Run #2). Using this process, it was found that approximately 15% more connection stiffness existed in the present PR frame when compared to the Xu and Grierson (1993) frame.

It is interesting to observe the similar total column weights considering that the columns resulting from the proposed algorithm do not decrease in size with increasing story heights (*i.e.* telescope). Therefore, the difference in weight is contained within the beams. Upon review of the evolved configuration for the PR frame (Run #2), it appears the proposed algorithm was struggling to find a good combination of beams and connections. Larger beam sizes may allow smaller columns, however, the algorithm was not able to find this balance.

Table 5.6: Weight Comparison for Frame 3

PR Frame	Xu and Grierson (1993)	Proposed Design Algorithm
Column Weight	51,888 lbs.	52,800 lbs.
Beam Weight	28,080 lbs.	46,260 lbs.
Total Weight	79,968 lbs.	99,060 lbs.

The structural behavior of the fittest FR and PR frames was examined using the ultimate load response shown in Figure 5.32. The FR and PR frames have virtually identical responses at load levels less than or equal to the target ultimate load ($\gamma_u = 1.0$). However, the FR frame follows a more gradual nonlinear response to the ultimate load levels when compared to the PR frame. The response of the PR frame to the lateral load combinations (F2 and F3) appears abrupt (*i.e.* lacking significant lateral deflection up to the ultimate load). This may signify the formation of a beam mechanism prior to lateral sway instability.

**Figure 5.32:** Load Deformation Response for Frame 3

The influence of the constraint criteria throughout the evolution was considered using penalty convergence trajectories. Figure 5.33 and 5.34 present the number of violating individuals for each service and ultimate constraint for each generation in the evolutionary process. Although the figures appear very busy, they exhibit characteristics that support some of the interesting evolved configurations and structural behavior observed previously.

The serviceability criteria converge nicely to a relatively few number of violating individuals in the later generations with the exception of the vertical displacement of the beams for the PR frame. This illustrates a significant shift in the algorithm’s effort. It appears the large columns (continuing over the entire height of the frame) have caused the PR connected beams to become the “weakest link”.

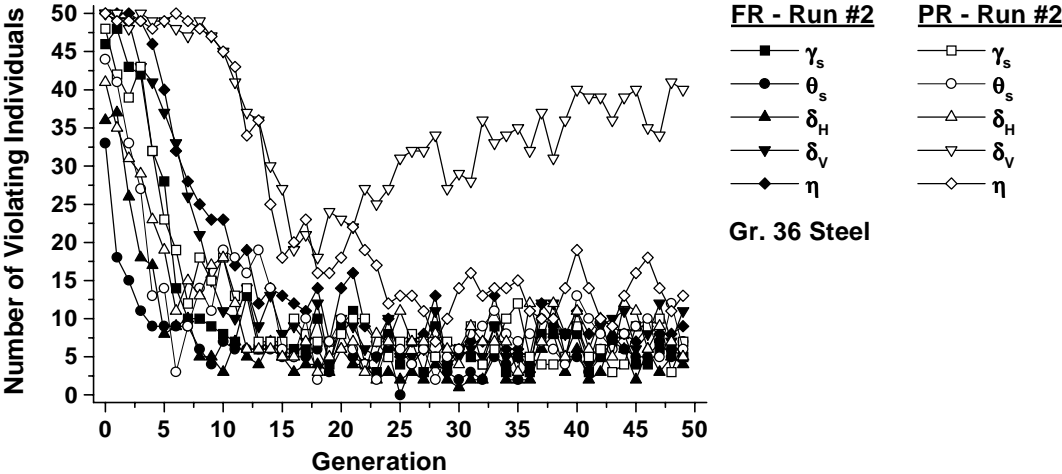


Figure 5.33: Service Penalty Convergence for Frame 3

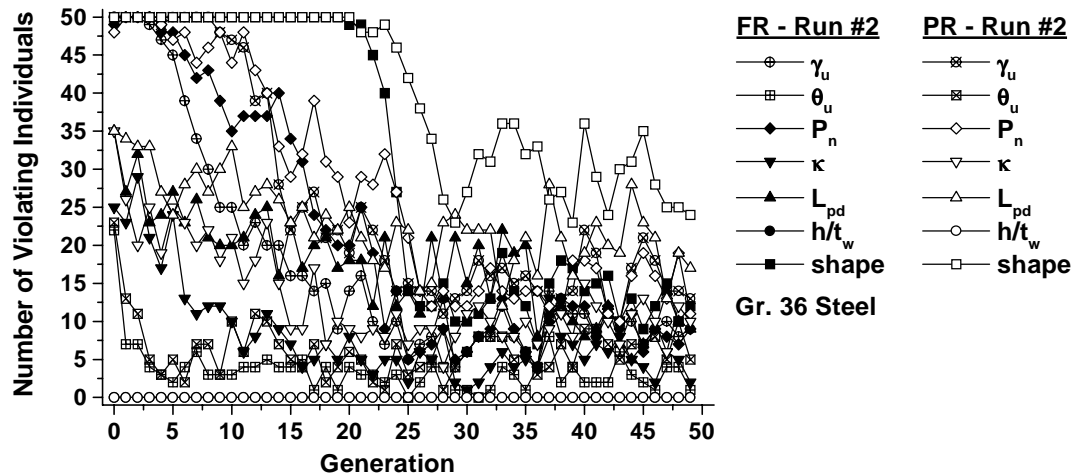


Figure 5.34: Ultimate Penalty Convergence for Frame 3

Contrary to the previous studies, there does not appear to be a predominant strength criterion controlling the design. The constructability constraint (shape) is dominant in all individuals for nearly the entire first half of the evolution for both PR and FR frames. However, several constraints are still active for individuals in the later generations with the exception of the web local buckling constraint that did not play a role in any of the results.

5.4 Concluding Remarks

This chapter presented the design of three frames using the proposed design algorithm. The results were compared to previous research found in the literature. The proposed algorithm provides reasonably proportioned designs for Frames 1 and 2. However, the designs for Frame 3 illustrated the algorithm's need for modification when designing large frames.

The algorithm consistently demonstrated stable convergence characteristics. This illustrates the ability of the proposed algorithm to identify and maintain good member sizes and connection types within the portions of the solution space provided via the randomly generated initial population and subsequent mutation. However, the lack of repeatability demonstrates the tendency of stochastic algorithms to converge to one of the multiple local minimums characteristic of the minimum weight design of steel frames. Furthermore, the considerable variation among the three runs performed for each frame configuration (as high as 20% for Frame 1, 25% for Frame 2, and 40% for Frame 3) introduces some doubt as to how close the evolved local minimum are to the global minimum (*i.e.* lightest frame).

The only way to determine the global minimum is to exhaustively search the entire solution space, however, this would require a prohibitively large population of individuals to consider all possible combinations of member sizes and connection stiffness. Appendix C provides the enumeration study for each of the previously studied frames. For grouped design variables and PR connections, the required population to exhaustively search the solution space is of the order (10^{17}) for Frame 1, (10^{13}) for frame 2, and (10^{64}) for Frame 3 (see Appendix C for FR connection calculations).

The predominant penalties found for each frame were the lateral-torsional buckling constraint and the constructability constraint limiting the column weight and size to be greater than or equal to the column it supports.

The results obtained will be used in the next chapter to offer recommendations for improvement of the proposed algorithm and suggestions for subsequent research endeavors.

Chapter 6 – Summary, Conclusions, and Future Work

The emphasis of this thesis was the development of an evolutionary design algorithm capable of providing “optimized” designs for unbraced steel frames using inelastic analysis as the design basis. The evolutionary algorithm provides a vehicle for incorporating inelastic analysis into an automated design algorithm suitable for performance-based design procedures. Furthermore, the evolutionary algorithm is a means of exploring the vast combinations of member sizes and connection types for frames with prescribed topology and loading by simulating the random reproduction (crossover and mutation) characteristics of the evolutionary process.

A secondary motivation of this research was to offer a first step towards the implementation of object-oriented (OO) programming to the design of structures using evolutionary computation. The representation of the design variables as objects not only is intuitive, but it also offers the potential for analytically modeling and evolving hybrid structures. As the performance of computers continues to rise (*i.e.* processing speeds in excess of 1400 MHz) and analytical tools continue to develop (*i.e.* 3-D analysis models and finite elements to model concrete/steel components), an automated, performance-based design tool for the design of hybrid structures becomes more and more realistic. Furthermore, an object-oriented evolutionary algorithm may be an ideal tool to realize this goal.

The purpose of this chapter is to summarize the previous chapters, draw some conclusions, and outline some ideas for future research.

6.1 Summary

The objective and scope of the thesis were introduced in Chapter 1. In addition, background and previous research were discussed for each of the three primary components of this research effort: optimization techniques, methods of advanced analysis, and object-oriented programming (OOP).

A review of the methods of optimization was presented in Chapter 2 in order to substantiate the use of the evolutionary algorithm. In addition, an illustrative example was provided to demonstrate the differences in representing the design variables as binary strings and as objects. Included in the example was a small parameter study to gain insight into the impact of object representation on the reproduction operations (crossover and mutation) and the ability of the object-oriented evolutionary algorithm (OO-EA) to find the optimal solution.

The advanced analysis based optimization problem was formulated in Chapter 3. First the inelastic algorithm, connection model, and assumptions of the distributed plasticity analysis were briefly discussed. Secondly, the objective function and constraint criteria were used to develop the unconstrained minimization problem. The constraint criteria include serviceability, strength, and constructability considerations.

A thorough description of the evolutionary design algorithm was presented in Chapter 4. The chapter begins with a brief review of the OOP, followed by a detailed description of each operation (*module*) within the evolutionary algorithm developed. The initialization *module* creates the initial population of frames. The evaluation *module* uses the results from the distributed plasticity analysis to assign fitness values (modified weight to account of connections and constraint violations) for each frame. The selection

module establishes a mating pool based on fitness using a roulette or tournament selection scheme. The reproduction *module* recombines the individuals (frames) in the mating pool via crossover (swapping of design variables) and mutation (changing of design variables).

Three frame studies were presented in Chapter 5 to demonstrate the application of the evolutionary design algorithm using frames found in the literature. Fitness trajectories and results from previous researchers were used to measure the performance of the evolutionary design algorithm created. Furthermore, the structural behavior of the evolved frames was investigated using load deformation-response curves. The influence of each constraint on the evolution was studied using penalty trajectories.

6.2 Conclusions

A number of conclusions can be drawn from this research effort. The following statements pertain to the evolutionary design algorithm.

1. The proposed algorithm provides well-proportioned designs for moderately sized frames (*e.g.* three-story, two-bay and two-story, three-bay frames). However, the designs of larger frames (*e.g.* a ten-story, three-bay frame) lack consistent beam sizes and connection types. In addition, the columns prematurely converge to uniform sizes for the entire height of the frame.
2. The lack of repeatability of the results of multiple design runs demonstrates the random nature of stochastic algorithms and illustrates the presence of multiple local minimums in the solution space.

3. The fitness trajectories demonstrate the ability of the proposed algorithm to identify the good “genetic” material within the population of frames and achieve stable convergence arriving at a fit (*i.e.* light) frame.
4. The penalty trajectories show the large influence of the constructability (shape) constraint for taller frames.

A couple statements can be made with regard to the structural behavior of the frames studied in Chapter 5.

1. The constraint on the column length to prevent lateral-torsional buckling is a highly influential constraint criterion. It greatly impacts the Grade 50 steel frames compared to the Grade 36 steel frames for larger floor-to-floor heights (*i.e.* greater than 12 feet) resulting in larger columns.
2. The web local buckling criterion according to AISC-LRFD is the least influential constraint criteria. In fact, it did not make a single contribution in the studies.

6.3 Recommendations for Future Work

The purpose of this research was to foster automated, performance-based design using object representation with the anticipation of broader applications. This thesis provides the development and initial implementation of an evolutionary design algorithm; however, a considerable amount of effort is left to advance its performance and application. The following are recommendations for improved performance.

1. Promote user interaction by incorporating an interface to allow the user to adjust parameters during the evolutionary process. For example, if the

algorithm is prematurely converging on a particular size, diversity can be re-established by increasing the mutation rate of the beams.

2. Implement a more sophisticated selection scheme. Voss and Foley (1999) suggests a rank-based fitness scheme that is not limited to merely fitness, but also considers individual constraints in its selection.
3. Perform a thorough parameter study to establish standards or general rules-of-thumb for the design parameters. For example, a suggested number of generations could be related to the number of design variables. The effect of crossover and mutation on evolution could be evaluated. The results for Frame 3 (ten-story, three-bay frame) of Chapter 5 may have been improved if the mutation rate, population size, and number of generations were increased.
4. The advanced analysis is computationally intensive, so any improvement to reduce the number of analyses will considerably shorten the duration of the evolution. One improvement is to detect whether a frame has already been analyzed in a previous or current generation and reuse the results from the analysis in the subsequent generation(s).
5. Seeding the population with “good” initial design variables will help make the exploration and convergence of the algorithm more efficient. This can be done by preliminarily sizing the beams exclusively under gravity loading and the columns exclusively under lateral loading using a simplified (rule-of-thumb) procedure such as the portal or cantilever method.

References

- ABCB. (1998). *Australian Standard - Steel Structures (AS 4100, 1998)*, Australian Building Codes Board & Standards Australia, Strathfield NSW.
- AISC. (1993). *Load and Resistance Factor Design Specification for Structural Steel Buildings*, American Institute of Steel Construction, Chicago, IL.
- Al-Saadoun, S. S., and Arora, J. S. (1989). "Interactive design optimization of framed structures." *Journal of Computing in Civil Engineering.*, v 3(n 1), p 60-74.
- Al-Salloum, Y. A. (1995). "A Pseudo-fully Stressed Design Approach for Optimum Design of Steel Frames." *International Journal for Numerical Methods in Engineering*, 38(20), 3513-3527.
- Al-Salloum, Y. A., and Almusallam, T. H. (1995). "Optimality and Safety of Rigidly- and Flexibly-jointed Steel Frames." *Journal of Constructional Steel Research*, 35(2), 189-215.
- Ambler, S. W. (1995). *The Object Primer - The Application Developer's Guide to Object-Oriented*, SIGS Books, New York.
- Arora, J. S. (1997). "Guide to Structural Optimization." ASCE Manuals and Reports on Engineering Practice, ASCE, New York.
- Arora, J. S., and Huang, M.-W. (1996). "Discrete structural optimization with commercially available sections." *Structural Engineering/Earthquake Engineering*, 13(2), 105s-122s.
- Balling, R. J. (1991). "Optimal Steel Frame Design by Simulated Annealing." *Journal of Structural Engineering*, 117, 1780-1795.
- Balling, R. J. (1997). "Combinatorial Search." *Guide To Structural Optimization*, J. S. Arora, ed., American Society of Civil Engineers, New York, NY, 323-326.
- Barakat, M., and Chen, W.-F. (1990). "Practical Analysis of Semi-Rigid Frames." *Engineering Journal-American Institute of Steel Construction*, 27(2), 54-68.
- Biedermann, J. D. (1996). "Addressing Current Issues in Structural Design Software." *Journal of Computing in Civil Engineering*, 10(4), 286-293.
- Bjorhovde, R., Colson, A., and Brozzetti, J. (1990). "Classification System for Beam-to-Column Connections." *Journal of Structural Engineering*, 116(11), 3059-3076.

- Bridge, R. Q. "The Inclusion of the Effects of Imperfections in Probability-Based Limit States Design." *Proceedings of the 1998 Structural Engineering World Congress*, San Francisco, CA, 9 pages.
- Bridge, R. Q., and Bizzanelli, P. "Imperfections in Steel Structures." *1997 Annual Technical Session and Meeting*, Toronto, ONT, 447-458.
- Camp, C., Pezeshk, S., and Cao, G. "Design of Framed Structures Using a Genetic Algorithm." *ASCE Structures Congress*, Chicago, IL, 19-30.
- Camp, C., Pezeshk, S., and Cao, G. (1998). "Optimized Design Of Two-Dimensional Structures Using A Genetic Algorithm." *Journal Of Structural Engineering*(May), 551-559.
- CEN. (1993). *Euro Code 3: Design of Steel Structures. Part 1-1: General Rules and Rules for Buildings (ENV 1993-1-1)*, European Committee for Standardization, Brussels.
- Chan, C. M. (1997). "How to Optimize Tall Steel Building Frameworks." *Guide To Structural Optimization*, J. S. Arora, ed., American Society of Civil Engineers, New York, NY, 93-120.
- Chan, C.-M., Grierson, D. E., and Sherbourne, A. N. (1995). "Automatic optimal design of tall steel building frameworks." *Journal of Structural Engineering-ASCE*, 121(5), 838-847.
- Chen, W. F., and Seung-Eock., K. (1997). *LRFD Steel Design using Advanced Analysis*, CRC Press, Inc.
- Clarke, M. J., and Bridge, R. Q. "Notional Load Approach for In-Plane Column Strength in Unbraced Frames." *Restructuring: America and Beyond: Proceedings of Structures Congress XIII*, Boston, MA, 1789-1792.
- Clarke, M. J., Bridge, R. Q., Hancock, G. J., and Trahair, N. S. (1992). "Advanced Analysis of Steel Building Frames." *Journal of Constructional Steel Research*, 23(1-3), 1-29.
- Dixon, A. S., and O'Brien, E. J. (1994). "Optimal Plastic Design of Steel Frames for Multiple Loadings." *Advances in Structural Engineering Computing International Conference on Computational Structures Technology*, CIVIL-COMP Ltd., Edinburgh, Scotland, 21-26.
- Ellingwood, B. (1989). "Serviceability Guidelines for Steel Structures." *Engineering Journal*, 26(1), 1-8.

- Erbatur, F., and Al-Hussainy, M. M. (1992). "Optimum Design of Frames." *Computers & Structures*, 45(5-6), 887-891.
- Erbatur, F., Hasancebi, O., Tutuncu, I., and Kilic, H. (2000). "Optimal design of planar and space structures with genetic algorithms." *Computers & Structures*, 75(2), 209-224.
- Foley, C. M. (1996). "Inelastic Behavior of Partially Restrained Steel Frames Using Parallel Processing and Supercomputers," Ph.D. Thesis, Marquette University, Milwaukee.
- Foley, C. M., and Vinnakota, S. (1995). "Toward Design Office Moment-Rotation Curves for End-Plate Beam-to-Column Connections." *Journal of Constructional Steel Research*, 8, 217-253.
- Foley, C. M., and Vinnakota, S. (1997). "Inelastic Analysis of Partially Restrained Unbraced Steel Frames." *Engineering Structures*, 19(11), 891-902.
- Foley, C. M., and Vinnakota, S. (1999a). "Inelastic Behavior of Multi-Story Partially Restrained Steel Frames - Part I." *Journal of Structural Engineering*, 125(8), 854-861.
- Foley, C. M., and Vinnakota, S. (1999b). "Inelastic Behavior of Multi-Story Partially Restrained Steel Frames - Part II." *Journal of Structural Engineering*, 125(8), 862-869.
- Garrett, J. H., and Hakim, M. M. (1992). "Object-Oriented Model of Engineering Design Standards." *Journal of Computing in Civil Engineering*, 6(3), 323-347.
- Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, New York.
- Goldberg, D. E., and Samtani, M. P. "Engineering Optimization via Genetic Algorithm." *Ninth Conference on Electronic Computation*, 471-482.
- Grierson, D. E. (1997a). "How to Optimize Structural Steel Frameworks." *Guide To Structural Optimization*, J. S. Arora, ed., American Society of Civil Engineers, New York, NY, 139-162.
- Grierson, D. E. (1997b). "An Optimality Criterion Method for Structural Optimization." *Guide To Structural Optimization*, J. S. Arora, ed., American Society of Civil Engineers, New York, NY, 303-314.
- Grierson, D. E., and Chan, C. M. (1993). "Optimality criteria design method for tall steel buildings." *Advances in Engineering Software*, 16(2), 119-125.

- Hager, K., and Balling, R. (1988). "New Approach For Discrete Structural Optimization." *Journal of Structural Engineering-ASCE*, 114(5), 1120-1134.
- Hajjar, J. F. (1997). "Effective Length and Notional Load Approaches for Assessing Frame Stability: Implications for American Steel Design." Task Committee on Load and Resistance Factor Design, ASCE.
- Hall, S. K., Cameron, G. E., and Grierson, D. E. (1989). "Least-weight design of steel frameworks accounting for P- Delta effects." *Journal of Structural Engineering-ASCE*, 115(6), 1463-1475.
- Hayalioglu, M. S. (2000). "Optimum Design of Geometrically Non-Linear Elastic-Plastic Steel Frames Via Genetic Algorithm." *Computers & Structures*, 77, 527-538.
- Hayalioglu, M. S., and Saka, M. P. (1992). "Optimum Design of Geometrically Non-Linear Elastic-Plastic Steel Frames with Tapered Members." *Computers & Structures*, 44(4), 915-924.
- Holland, J. H. (1975). *Adaptation In Natural And Artificial Systems, (An Introductory Analysis With Applications To Biology, Control, and Artificial Intelligence)*, MIT Press, Cambridge, London.
- Jenkins, W. M. (1991). "Towards structural optimization via the genetic algorithm." *Computers and Structures*, 40(5), 1321-1327.
- Jenkins, W. M. (1992). "Plane Frame Optimum Design Environment Based on Genetic Algorithm." *Journal of Structural Engineering*, 118(11), 3103-3112.
- Jingui, L., Yunliang, D., Bin, W., and Shide, X. (1996). "An Improved Strategy for GAs in Structural Optimization." *Computers & Structures*, 61(6), 1185-1191.
- Kameshki, E. S., and Saka, M. P. (1999). "Optimum Design of Nonlinear Steel Frames with Semi-rigid Connections using Genetic Algorithms." *Optimization and Control in Civil and Structural Engineering*, B. H. V. Topping and S. Kumar, eds., CIVIL-COMP Ltd., Edinburgh, Scotland, 95-105.
- Kim, S.-E., and Chen, W.-F. "Practical Advanced Analysis for Steel Frame Design." *1996 12th Structures Congress*, Chicago, IL, 19-30.
- Kim, S.-E., and Chen, W.-F. (1996b). "Practical Advanced Analysis for Unbraced Steel Frame Design." *Journal of Structural Engineering*, 122(11), 1259-1265.
- Kishi, N., and Chen, W.-F. (1990). "Moment-Rotation Relations of Semi-Rigid Connections with Angles." *Journal of Structural Engineering*, 116(7), 1813-1834.

- Leite, J. P. B., and Topping, B. H. V. (1998). "Improved Genetic Operators for Structural Engineering Optimization." *Advances in Engineering Software*, 29(7-9), 529-562.
- Liebman, J. S., Khachaturian, F., Chanaratna, V. (1981). "Discrete Structural Optimization." *Journal of the Structural Division, ASCE*, 107(ST11), 2177-2197.
- Liew, J.-Y. R., and Chen, W.-F. (1993a). "Second-Order Refined Plastic-Hinge Analysis for Frame Design - Part I." *Journal of Structural Engineering*, 119(11), 3196-3216.
- Liew, J.-Y. R., and Chen, W.-F. (1993b). "Second-Order Refined Plastic-Hinge Analysis for Frame Design - Part II." *Journal of Structural Engineering*, 119(11), 3217-3237.
- Liew, J.-Y. R., White, D. W., and Chen, W.-F. (1993a). "Limit States Design of Semi-Rigid Frames Using Advanced Analysis - Part I." *Journal of Constructional Steel Research*, 26(1), 1-29.
- Liew, J.-Y. R., White, D. W., and Chen, W.-F. (1993b). "Limit States Design of Semi-Rigid Frames Using Advanced Analysis - Part II." *Journal of Constructional Steel Research*, 26(1), 29-57.
- Lutz, M., and Ascher, D. (1999). *Learning Python*, O'Reilly & Associates, Inc., Sebastopol, CA.
- Maleck, A., and White, D. W. "Effects of Geometric Imperfections on Steel Framing Systems." *1998 SSRC Annual Technical Session*, Atlanta, GA, xxx-xxx.
- May, S. A., and Balling, R. J. "Strategies which permit multiple discrete section properties per member in 3D frameworks." *10th Conference on Electronic Computation*, p 189-196.
- May, S. A., and Balling, R. J. (1992). "A Filtered Simulated Annealing Strategy for Discrete Optimization of 3D Steel Frameworks." *Structural Optimization*, 4, 142-148.
- Pezeshk, S. (1997). "How to Optimize Frames using Plastic Design Concept." *Guide To Structural Optimization*, J. S. Arora, ed., American Society of Civil Engineers, New York, NY, 197-209.
- Pezeshk, S., Camp, C. V., and Chen, D. "Optimal Design of 2-D Frames Using a Genetic Algorithm." *International Workshop on Optimal Performance of Civil Infrastructure Systems: Structures Congress*, Portland, OR, 155-168.

- Pezeshk, S., Camp, C. V., and Chen, D. (2000). "Design of Nonlinear Framed Structures Using Genetic Optimization." *Journal of Structural Engineering*, 126(3), 382-388.
- Rajan, S. D. (1995). "Sizing, Shape, and Topology Design Optimization of Trusses Using Genetic Algorithm." *Journal of Structural Engineering*, 121(10), 1480-1487.
- Rajan, S. D. (2001). *Introduction to Structural Analysis & Design*, John Wiley & Sons, Inc.
- Rajeev, S., and Krishnamoorthy, C. S. (1992). "Discrete Optimization of Structures Using Genetic Algorithms." *Journal of Structural Engineering*, 118(5), 1233-1250.
- Rajeev, S., and Krishnamoorthy, C. S. (1997). "Genetic Algorithms-Based Methodologies for Design Optimization of Trusses." *Journal of Structural Engineering*, 123(3), 350-358.
- Rigopoulos, D. R., and Oppenheim, I. J. (1992). "Intelligent Objects for Synthesis of Structural Systems." *Journal Computing in Civil Engineering*, 6(3), 266-281.
- Rivard, H., and Fenves, S. J. (2000). "A Representation for Conceptual Design of Buildings." *Journal of Computing in Civil Engineering*, 14(3), 151-159.
- Saka, M. P. (1991). "Optimum Design of Steel Frames with Stability Constraints." *Computers & Structures*, 41(6), 1365-1377.
- Saka, M. P., and Hayalioglu, M. S. (1991). "Optimum Design of Geometrically Nonlinear Elastic-Plastic Steel Frames." *Computers & Structures*, 38(3), 329-344.
- Sause, R., Martini, K., and Powell, G. H. (1992). "Object-Oriented Approaches for Integrated Engineering Design Systems." *Journal of Computing in Civil Engineering*, 6(3), 248-265.
- Shrestha, S. M., and Ghaboussi, J. (1998). "Evolution of Optimum Structural Shapes Using Genetic Algorithm." *Journal of Structural Engineering-ASCE*, 124(11), 1331-1338.
- Simoës, L. M. C. (1996). "Optimization of Frames with Semi-Rigid Connections." *Computers & Structures*, 60(4), 531-539.
- SODA. (1999). "SODA - Structural Optimization Design Analysis.", Acronym Software, Inc., Waterloo, ONT.

- Spears, W. M., Jong, K. A. D., Back, T., Fogel, D. B., and Garis, H. d. "An Overview of Evolutionary Computation." *6th European Conference on Machine Learning*, 442-459.
- SSRC. (1988). "Technical Memorandum Number 5." *Guide to Stability Design Criteria for Metal Structures, 4th Edition*, T. V. Galambos, ed., John Wiley & Sons, New York, New York.
- Thevendran, V., Das Gupta, N. C., and Tan, G. H. (1992). "Minimum Weight Design of Multi-bay Multi-storey Steel Frames." *Computers & Structures*, 43(3), 495-503.
- Voss, M. S. (1992). "Object Oriented Modeling of the 1991 National Design Specification for Wood Construction Interaction Equations," MS Thesis, Marquette University, Milwaukee, WI.
- Voss, M. S., and Foley, C. M. "An Evolutionary Algorithm for Structural Optimization." *1999 Genetic and Evolutionary Algorithm Conference (GECCO 1999)*, Orlando, FL, 675-688.
- Voss, M. S., and Foley, C. M. "The μ - λ α - β Distribution: A Selection Scheme for Ranked Populations." *Late Breaking Papers: 1999 Genetic and Evolutionary Algorithm Conference (GECCO 1999)*, Orlando, FL, 675-688.
- White, D. W. (1992). "Plastic-Hinge Based Methods for Frame Design." *Journal of Constructional Steel Research*, 24, 121-152.
- White, D. W. (1993). "Plastic-Hinge Methods for Advanced Analysis of Steel Frames." *Journal of Constructional Steel Research*, 24(2), 121-152.
- White, D. W., and Nukala, P. K. V. N. "Recent Advances in Methods for Inelastic Frame Analysis: Implications for Design and a Look Toward the Future." *Proceedings of the National Steel Construction Conference*, Chicago, IL, 43:1-24.
- Whitley, D. "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best." *Proceedings Of The Third International Conference On Genetic Algorithms*, George Mason University, 116-121.
- WoodWorks. (2000). "WoodWorks Design Office.", American Forest and Paper Association, Canadian Wood Council.
- Wu, A. S., Lindsay, R. K., and Riolo, R. L. "Emprical Observations on the Roles of Crossover and Mutation." *Proceedings of the Seventh International Conference on Genetic Algorithms*, San Francisco, CA, pp. 362-369.

- Xu, L., and Grierson, D. E. (1993). "Computer-automated design of semirigid steel frameworks." *Journal of Structural Engineering-ASCE*, 119(6), 1740-1760.
- Xu, L., Sherbourne, A. N., and Grierson, D. E. (1995). "Optimal Cost Design of Semi-Rigid, Low-Rise Industrial Frames." *Engineering Journal*(Third Quarter), 87-97.
- Yang, J., and Soh, C. K. (1997). "Structural optimization by genetic algorithms with tournament selection." *Journal of Computing in Civil Engineering*, 11(3), 195-200.
- Yura, J. A., Galambos, T. V., and Ravindra, M. K. (1978). "The Bending Resistance of Steel Beams." *Journal of the Structural Division*, 104(ST9), 1355-1370.
- Ziemian, R. D. (1992). "Advanced Methods of Inelastic Analysis in the Limit States of Steel Structures," Ph.D. Dissertation, Cornell University, Ithaca, NY.
- Ziemian, R. D., McGuire, W., and Deierlein, G. G. (1992a). "Inelastic Limit States Design. Part I - Planar Frame Studies." *Journal of Structural Engineering*, 118(9), 2532-2549.
- Ziemian, R. D., McGuire, W., and Deierlein, G. G. (1992b). "Inelastic Limit States Design. Part II - Planar Frame Studies." *Journal of Structural Engineering*, 118(9), 2550-2568.

Appendix A
Source Code for the OO-EA used in Illustrative
Example (Chapter 2)

```

#####
# MODULE: 'oo-ea'.py
#
# Module to run and evaluate the evolutionary algorithm using the algebraic function example from Jenkins, (1991)
#
# Open Source Python ver. 1.5
#####

from RandomArray import*
import copy

class population:
    def __init__(self,popSize,numParams,lb,ub):
        self.indiv = []
        self.fit = []
        self.lb = lb
        self.ub = ub
        self.numParams = numParams
        for i in range(popSize):
            self.indiv.append(individual(self))

    def fitness(self):
        self.fit = []
        tmpfit = []
        self.fitList = []
        for i in range(len(self.indiv)):
            val1 = self.indiv[i].value[0]
            val2 = self.indiv[i].value[1]
            C = 0.75*self.ub**2.0 + self.ub*self.ub + 1.25*self.ub**2.0
            if val1 + val2 < 8.0:
                y = C
            else:
                y = 0.75*val1**2.0 + val1*val2 + 1.25*val2**2.0
            tmpfit.append(C-y,i)
        fsum = 0.0
        for i in range(len(self.indiv)):
            fsum = fsum + tmpfit[i][0]
        favg = fsum/len(self.indiv)
        fmax = max(tmpfit)[0]

```

```

if favg == fmax:
    print('convergence')
    for i in range(len(self.indiv)):
        print(i,self.indiv[i].value)
for i in range(len(self.indiv)):
    if tmpfit[i][0] == 0.0:
        tmp = 0.0
    else:
        tmp = (favg/(fmax-favg))*tmpfit[i][0] + favg*(1.0-(favg/(fmax-favg)))
        if tmp < 0.0:
            tmp = 0.0
        self.fit.append(int(tmp),i)
        self.fitList.append(int(tmp),i)
self.fitList.sort()
self.bestIndiv = copy.deepcopy(self.indiv[self.fitList[len(self.indiv)-1][1]])

def selection(self):
    numcopies = []
    matingPool = []
    newPop     = []
    sumFit     = 0.0
    for i in range(len(self.indiv)):
        sumFit = sumFit + self.fit[i][0]
    for i in range(len(self.indiv)):
        tmp = len(self.indiv)*self.fit[i][0]/sumFit
        if tmp - int(tmp) < 0.5:
            rnd = 0.0
        else:
            rnd = 1.0
        ncopy = int(tmp) + rnd
        numcopies.append(ncopy,i)
    for i in range(len(self.indiv)):
        for j in range(numcopies[i][0]): # skips over if number of copies is zero (i.e. for i in range(0))
            matingPool.append(i) # appends indiv num
    if len(matingPool) >= len(self.indiv):
        for i in range(len(self.indiv)):
            newPop.append(copy.deepcopy(self.indiv[matingPool[i]]))

```

```

else:
    for i in range(len(matingPool)):
        newPop.append(copy.deepcopy(self.indiv[matingPool[i]]))
    diff = len(self.indiv) - len(newPop)
    for i in range(diff):
        ranNum = randint(0,len(matingPool))
        newPop.append(copy.deepcopy(self.indiv[matingPool[ranNum]]))
self.indiv = []
for i in range(len(newPop)):
    self.indiv.append(newPop[i])

def xover(self,rate,nonHomRate):
    nextGen = []
    for ii in range(len(self.indiv)):      # temporarily store the old generation
        nextGen.append(copy.deepcopy(self.indiv[ii]))
    for i in range(len(self.indiv)):
        if random() <= rate:
            B = randint(0,len(self.indiv))
            if i == B:
                if i == len(self.indiv)-1:
                    B = B - 1
                else:
                    B = B + 1
            if random() <= nonHomRate:
                j = randint(0,self.numParams)
                if j == 0:
                    nextGen[i].value[j] = self.indiv[B].value[j+1]
                else:
                    nextGen[i].value[j] = self.indiv[B].value[j-1]
            else:
                j = randint(0,self.numParams)
                nextGen[i].value[j] = self.indiv[B].value[j]
    self.indiv = []
    for ii in range(len(nextGen)):      # temporarily store the old generation
        self.indiv.append(copy.deepcopy(nextGen[ii]))

def display(self,genNum):
    print('\n')
    for i in range(len(self.indiv)):
        print(genNum,i,self.indiv[i].value,self.fit[i])

```

```

def mutation(self,rate):
    for i in range(len(self.indiv)):
        if random() <= rate:
            A = randint(0,len(self.indiv))
            B = randint(0,self.numParams)
            self.indiv[A].value[B] = randint(self.lb,self.ub+1)

def elitism(self):
    A = randint(0,len(self.indiv))
    self.indiv[A] = copy.deepcopy(self.bestIndiv)

class individual:
    def __init__(self,pop):
        self.value = []
        for i in range(pop.numParams):
            self.value.append(randint(pop.lb,pop.ub+1))

#----OO-EA
genNum = 0
genMax = 50
#----Initialize-(popSize,number of parameters,lower bound,upper bound)
pop1 = population(20,2,0,15)
#----Fitness
pop1.fitness()
Evolutionary Algorithm:
while genNum <= genMax:
    #----Display
    pop1.display(genNum)
    #----Select
    pop1.selection()
    #----Crossover-(rate,nonhomologous rate)
    pop1.xover(0.60,0.0)
    #----Mutate-(rate)
    pop1.mutation(1.0)
    #----Elitism
    pop1.elitism()
    #----Fitness
    pop1.fitness()
    genNum = genNum + 1

```

Appendix B
Source Code for the Proposed EA (Chapter 4)

```

#####
# MODULE: 'frame'.py
#
# Module to run the evolutionary algorithm for the "optimized" design of unbraced steel FRAMES
#
# Open Source Python ver. 1.5
#####

import initialization
import evaluation
import selection
import reproduction

#-----numStories,firStryHt,typStryHt,numBays,extBay,intBay,bayWidth,E,Fy,connType,varType
bldgInfo = [2,15.0,12.0,3,20.0,20.0,20.0,29000000.0,36000.0,'FR','grp']
#-----wind(psf),floorDL(psf),roofDL(psf),floorLL(psf),roofLL(psf)
loadInfo = [20.0,50.0,15.0,75.0,50.0]
#-----noSubCol,noSubBm,lodFacInc,numLoadCases
evalInfo = [10,10,0.05,6]
#-----latSway(L/#),vertDef(L/#),%Yld(#),Curve(*yldStrain)
limitInfo = [400,360,15,4]

popSize = 50
genMax = 50
genNum = 1

#----INIALIZE
pop1 = initialization.population(bldgInfo,popSize)
#----EVALUATE
evaluation.advAnalysis(pop1,evalInfo,loadInfo)
PHIs = evaluation.penalty(pop1,limitInfo)
#----FITNESS
pop1.fitness(PHIs)
#----DISPLAY
pop1.display('display/display' + '0' + '.txt')
#----REPORT
pop1.report(PHIs,genNum,'report/report'+ '0' + '.txt')

```

```
#----EVOLUTIONARY ALGORITHM
while genNum <= genMax:
    #----SELECT
    selection.tournament(pop1,PHIs,2,0.40,0.70)
    #----REPRODUCE
    reproduction.crossover(pop1,0.60,0.60,0.001)
    #----MUTATE
    reproduction.mutate(pop1,0.001,0.001,0.30,0.30,0.001)
    #----ELITISM
    reproduction.elitism(pop1)
    #----EVALUATE
    evaluation.advAnalysis(pop1,evalInfo,loadInfo)
    PHIs = evaluation.penalty(pop1,limitInfo)
    #----FITNESS
    pop1.fitness(PHIs)
    #----DISPLAY
    pop1.display('display/display'+`genNum`+'.txt')
    #----REPORT
    pop1.report(PHIs,genNum,'report/report'+`genNum`+'.txt')
    genNum = genNum + 1
```

```

#####
# MODULE: initialization.py
#
# Module to INITIALIZE a population of frames and establish individual weights & fitness
#
# Open Source Python ver. 1.5
#####

import string
from Numeric import *
from RandomArray import *
import aisc
import cdata

class population:
    def __init__(self,bldgInfo,popSize):
        self.dbBeams = aisc.readWshapes('beam')
        self.dbColumns = aisc.readWshapes('column')
        self.dbConn = cdata.readConnData(5)
        self.colList = aisc.makeColList(self.dbColumns,'dn','bf/2tf')
        self.bmList = aisc.makeBmList(self.dbBeams,'dn','bf/2tf','h/tw')
        self.numStories = bldgInfo[0]
        self.firStryHt = bldgInfo[1]
        self.typStryHt = bldgInfo[2]
        self.numBays = bldgInfo[3]
        self.extBay = bldgInfo[4]
        self.intBay = bldgInfo[5]
        self.bayWidth = bldgInfo[6]
        self.E = bldgInfo[7]
        self.Fy = bldgInfo[8]
        self.connType = bldgInfo[9]
        self.varType = bldgInfo[10]
        self.building = []
        self.wt = []
        self.fit = []
        for i in range(popSize):
            self.building.append(building(self))
            self.building[i].setStories(self)

```

```

def fitness(self,PHI):
    self.fit = []
    self.wt = []
    fitFeas = []
    for i in range(len(self.building)):
        bldgWt = self.building[i].weight(self)
        self.wt.append(bldgWt,i)
    for i in range(len(self.building)):
        compProduct = 1
        for j in range(len(PHI)):
            compProduct = compProduct * PHI[j][i][0]
        compSum = compProduct * self.wt[i][0]
        self.fit.append((compSum,i))

def display(self,filename):
    output = open(filename,'w')
    for i in range(len(self.building)):
        output.write('%s %i\t%s %.4e\t%s %.4e \n'%( 'building',i,'fit',self.fit[i][0],'wt',self.wt[i][0]))
        for j in range(self.numStories):
            col = ''
            bm = '\t'
            colHt = ''
            stryHt = ''
            for k in range(self.numBays+1):
                colTmp = self.building[i].story[self.numStories-j-1].column[k].shape
                colHtTmp = 'c%s'%(self.building[i].story[self.numStories-j-1].column[k].L)
                stryHtTmp = 's%s'%(self.building[i].story[self.numStories-j-1].stryHt)
                col = col + colTmp + '\t\t\t'
                colHt = colHt+colHtTmp + '\t\t\t'
                stryHt = stryHt + stryHtTmp + '\t\t\t'
            for m in range(self.numBays):
                bmTmp = self.building[i].story[self.numStories-j-1].beam[m].shape
                lcTmp = self.building[i].story[self.numStories-j-1].beam[m].connection[0].type
                rcTmp = self.building[i].story[self.numStories-j-1].beam[m].connection[1].type
                bm = bm + lcTmp + ' ' + bmTmp + ' ' + rcTmp + '\t\t'
            output.write('\n' + bm + '\n' + col + '\n' + colHt + '\n' + stryHt)
        output.write('\n' + '_____ ' + '\n')

```



```

        wtFact0 = pop.dbConn[self.story[j].beam[m].connection[0].type][2][0]
        wtFact1 = pop.dbConn[self.story[j].beam[m].connection[1].type][2][0]
        bmWtTmp2 = (wtFact0 + wtFact1 - 1) * bmWtTmp1 * bmLength
    elif bmWtTmp1 > 35.0 and bmWtTmp1 <= 161.0:
        wtFact0 = pop.dbConn[self.story[j].beam[m].connection[0].type][2][1]
        wtFact1 = pop.dbConn[self.story[j].beam[m].connection[1].type][2][1]
        bmWtTmp2 = (wtFact0 + wtFact1 - 1) * bmWtTmp1 * bmLength
    else:
        wtFact0 = pop.dbConn[self.story[j].beam[m].connection[0].type][2][2]
        wtFact1 = pop.dbConn[self.story[j].beam[m].connection[1].type][2][2]
        bmWtTmp2 = (wtFact0 + wtFact1 - 1) * bmWtTmp1 * bmLength
    bmWt = bmWt + bmWtTmp2
    bldgWt = bldgWt + colWt + bmWt

    return bldgWt

```

```
class story:
```

```

    def __init__(self,stryHt):
        self.stryHt = stryHt
        self.beam = []
        self.column = []

```

```

    def setIndivSizes(self,pop):
        for m in range(pop.numBays):
            self.beam.append(beam())
            self.beam[m].setSize(pop)
            if pop.connType == 'PR1' or pop.connType == 'FR':
                tmpCon = connection()
                tmpCon.setType(pop)
                for n in range(2):
                    self.beam[m].connection.append(tmpCon)
            else:
                for n in range(2):
                    self.beam[m].connection.append(connection())
                    self.beam[m].connection[n].setType(pop)
        for k in range(pop.numBays+1):
            self.column.append(column(self.stryHt))
            self.column[k].setSize(pop)

```

```

# Note: the def setIndivSizes method instantiates different objects (allocates different memory addresses)
#       to each component (column, beam, connection) of the frame

```

```

def setGrpSizes(self, pop):
    grpBm = beam()
    grpBm.setSize(pop)
    for m in range(pop.numBays):
        self.beam.append(grpBm)
    grpCon = connection()
    grpCon.setType(pop)
    for n in range(2):
        self.beam[m].connection.append(grpCon)
    grpExtCol = column(self.stryHt)
    grpExtCol.setSize(pop)
    grpIntCol = column(self.stryHt)
    grpIntCol.setSize(pop)
    self.column.append(grpExtCol)
    for k in range(1, pop.numBays):
        self.column.append(grpIntCol)
    self.column.append(grpExtCol)

```

Note: the def setGrpSizes method instantiates one object (allocates the same memory address) to a group
of similar components (beams, connections, interior columns, and exterior columns) for each story

```

def weight(self, pop):
    colWt = 0
    bmWt = 0
    for k in range(pop.numBays+1):
        colWtTmp = pop.dbColumns[self.column[k].shape]['plf'] * self.column[k].L
        colWt = colWt + colWtTmp
    for m in range(pop.numBays):
        if m == 0 or m == pop.numBays-1:
            bmLength = pop.extBay
        else:
            bmLength = pop.intBay
        bmWtTmp1 = pop.dbBeams[self.beam[m].shape]['plf'] * bmLength
        bmWt = bmWt + bmWtTmp1
    storyWt = colWt + bmWt
    return storyWt

```

```

class beam:
    def __init__(self):
        self.connection = []

    def setSize(self,pop):
        bf2tfMax    = 65 / (pop.Fy / 1000) ** 0.50
        htwMax      = 640 / (pop.Fy / 1000) ** 0.50
        nomDpth     = self.getRandSize()
        Beams       = aisc.getBmList(pop.bmList,nomDpth,bf2tfMax,htwMax)
        if len(Beams) == 0:
            nomDpth = nomDpth + 2
            Beams   = aisc.getBmList(pop.bmList,nomDpth,self.bf2tfMax,self.htwMax)
            length  = len(Beams)
        else:
            length = len(Beams)
        shpKey      = randint(1,length)
        self.shape  = Beams[shpKey-1][4]

    def getRandSize(self):
        RanNum = random()
        if RanNum <= 0.10:
            nomSz = 12
        elif RanNum <= 0.20:
            nomSz = 14
        elif RanNum <= 0.30:
            nomSz = 16
        elif RanNum <= 0.40:
            nomSz = 18
        elif RanNum <= 0.50:
            nomSz = 21
        elif RanNum <= 0.60:
            nomSz = 24
        elif RanNum <= 0.70:
            nomSz = 27
        elif RanNum <= 0.80:
            nomSz = 30
        elif RanNum <= 0.90:
            nomSz = 33
        else:
            nomSz = 36
        return nomSz

```

```

class column:
    def __init__(self,stryHt):
        self.L = stryHt

    def setSize(self,pop):
        bf2tfMax = 65 / (pop.Fy / 1000) ** 0.50
        nomDpth = self.getRandSize()
        Columns = aisc.getColList(pop.colList,nomDpth,bf2tfMax)
        if len(Columns) == 0:
            nomDpth = nomDpth + 2
            Columns = aisc.getColList(pop.colList,nomDpth,bf2tfMax)
            length = len(Columns)
        else:
            length = len(Columns)
        shpKey = randint(1,length)
        self.shape = Columns[shpKey-1][3]

    def getRandSize(self):
        RanNum = random()
        if RanNum <= 0.25:
            nomSz = 8
        elif RanNum <= 0.50:
            nomSz = 10
        elif RanNum <= 0.75:
            nomSz = 12
        else:
            nomSz = 14
        return nomSz

class connection:
    def __init__(self):
        self.type = ''

    def setType(self,pop):
        if pop.connType == 'FR':
            self.type = 'C1'
        else:
            self.type = self.getRandConnect()

```

```
def getRandConnect(self):
    ranNum = random()
    if ranNum <= 0.20:
        type = 'C1'
    elif ranNum <= 0.40:
        type = 'C2'
    elif ranNum <= 0.60:
        type = 'C3'
    elif ranNum <= 0.80:
        type = 'C4'
    else:
        type = 'C5'
    return type
```

```

#####
# MODULE: evaluation.py
#
# Module to EVALUATE the building population and PENALIZE individuals violating the constraints
#
# Open Source Python ver. 1.5.2
#####

import os
import string
from Numeric import *

def advAnalysis(pop,evalInfo,loadInfo):
    global noSubCol,noSubBm,lodFacInc,numLoadCases
    global wind,floorDL,roofDL,floorLL,roofLL
    noSubCol      = evalInfo[0]
    noSubBm       = evalInfo[1]
    lodFacInc     = evalInfo[2]
    numLoadCases  = evalInfo[3]
    wind          = loadInfo[0]
    floorDL       = loadInfo[1]
    roofDL        = loadInfo[2]
    floorLL       = loadInfo[3]
    roofLL        = loadInfo[4]
    getFileNames(pop)
    for j in range(len(pop.building)):
        for k in range(numLoadCases):
            if k <= 2:
                gamFlag = 3
                outFlag = 2
            else:
                gamFlag = 4
                outFlag = 1
            fileName = pop.building[j].outputFiles[k]
            output = open('evaluate/input.txt','w')
            output.write(fileName + '\n')
            writeBuildingData(pop,pop.building[j],k,output,fileName)
            output.write(`lodFacInc` + ' ' + `gamFlag` + ' ' + `outFlag` + '\n')
            output.close()
            os.chdir('c:\\EvDesign\\evaluate')
            os.system('evaluate.exe')
            os.chdir('c:\\EvDesign')

```

```

def getFileNames(pop):
    for i in range(len(pop.building)):
        pop.building[i].outputFiles = []
        if i <= 9:
            IndNumTmp = `0` + `i`
        else:
            IndNumTmp = `i`
        for loadCase in range(numLoadCases):
            if loadCase < 1:
                CseTmp = 's'
                NumTmp = '1'
            elif loadCase < 2:
                CseTmp = 's'
                NumTmp = '2'
            elif loadCase < 3:
                CseTmp = 's'
                NumTmp = '3'
            elif loadCase < 4:
                CseTmp = 'f'
                NumTmp = '1'
            elif loadCase < 5:
                CseTmp = 'f'
                NumTmp = '2'
            else:
                CseTmp = 'f'
                NumTmp = '3'
            fileName = IndNumTmp + CseTmp + NumTmp + '.txt'
            pop.building[i].outputFiles.append(fileName)

def writeBuildingData(pop,building,loadcase,output,fileName):
    NumMemb = (pop.numBays * pop.numStories) + (pop.numBays + 1) * pop.numStories
    NumJts = (pop.numBays + 1) * (pop.numStories + 1)
    NumRes = (pop.numBays + 1) * 3 # fixed base supports
    NumRjts = pop.numBays + 1
    outString = `NumMemb` + ` ` + `NumJts` + ` ` + `NumRes` + ` ` + `NumRjts` + ` ` + `pop.E`
    output.write(outString + `\\n`)
    Cntr = 0 # output the nodal coordinates
    Ybase = 0.0
    for ii in range(pop.numStories+1):
        for jj in range(pop.numBays+1):
            JtNum = jj + 1 + Cntr
            if jj == 0:

```

```

        Xcoord = 0.0 * 12.0
    elif jj == pop.numBays:
        Xcoord = (pop.extBay * 2 + pop.intBay*(pop.numBays - 2)) * 12.0
    else:
        Xcoord = (pop.extBay + (jj - 1) * pop.intBay) * 12.0
    Ycoord = Ybase * 12.0
    OutString = `JtNum` + ' ' + `Xcoord` + ' ' + `Ycoord`
    output.write(OutString + '\n')
if ii == 0:
    Ybase = Ybase + pop.firStryHt
else:
    Ybase = Ybase + pop.typStryHt
Cntr = Cntr + jj + 1
for ii in range(pop.numStories):
    ColCtr = pop.numBays + 1
    for kk in range(pop.numBays+1):          # column connectivity
        MemNum = kk + 1 + ii * (2 * pop.numBays + 1)
        LeftEnd = kk + 1 + ii * ColCtr
        RightEnd = kk + 1 + pop.numBays + 1 + ii * ColCtr
        MemFlag = 0                          # column Flag
        OutString = `MemNum` + ' ' + `LeftEnd` + ' ' + `RightEnd` + ' ' + `MemFlag`
        output.write(OutString + '\n')
    BmCtr = MemNum
    for kk in range(pop.numBays):          # beam connectivity
        MemNum = kk + BmCtr + 1
        LeftEnd = kk + pop.numBays + 1 + ii * (pop.numBays + 1) + 1
        RightEnd = LeftEnd + 1
        MemFlag = 1 # Beam Flag
        OutString = `MemNum` + ' ' + `LeftEnd` + ' ' + `RightEnd` + ' ' + `MemFlag`
        output.write(OutString + '\n')
for storyNum in range(pop.numStories):    # write out story (member) information
    writeStoryData(pop,output,building.story[storyNum],storyNum)
for ii in range(pop.numBays+1):          # write restraint information
    ResJtNum = ii + 1
    OutString = `ResJtNum` + ' ' + '1' + ' ' + '1' + ' ' + '1'
    output.write(OutString + '\n')
chkString = fileName[2:4]                # output loading parameters
if chkString == 's1':                    # set up load factors
    factors = [1.0,0.80,0.8,0.0]
elif chkString == 's2':
    factors = [1.0,0.4,0.4,0.5]

```

```

elif chkString == 's3':
    factors = [1.0,0.4,0.4,-0.5]
elif chkString == 'f1':
    factors = [1.2,1.6,0.5,0.0]
elif chkString == 'f2':
    factors = [1.2,0.5,0.5,1.3]
else:
    factors = [1.2,0.5,0.5,-1.3]
if chkString != 's1':
    # lateral & Gravity Loads.'f1' includes notional loads.
    if pop.numStories == 1:
        # only one story - No gravity nodal loads from upper cols.
        NumLoadJt = 1
        # notional (horz.) & Gravity (vert.-from upper columns)
    else:
        NumLoadJt = pop.numStories + (pop.numStories - 1) * (pop.numBays + 1)
else:
    # gravity ONLY
    if pop.numStories == 1:
        # only one story - No gravity nodal loads from upper cols.
        NumLoadJt = 0
        # notional (horz.) & Gravity (vert.-from upper columns)
    else:
        NumLoadJt = (pop.numStories - 1) * (pop.numBays + 1)
NumLoadMem = pop.numBays * pop.numStories
OutString = `NumLoadJt` + ' ' + `NumLoadMem`
output.write(OutString + '\n')
if chkString != 's1':
    # lateral & Gravity Loads.'f1' includes notional loads.
    writeLatLoads(pop,building,output,factors,chkString)
    writeGravLoads(pop,building,output,factors,chkString)
else:
    # gravity ONLY
    writeGravLoads(pop,building,output,factors,chkString)

def writeStoryData(pop,output,story,storyNum):
    ColCtr = pop.numBays + 1
    for kk in range(pop.numBays+1):
        MemNum = kk + 1 + storyNum * (2 * pop.numBays + 1)
        writeColumnData(pop,story.column[kk],output,MemNum)
    BmCtr = MemNum
    for kk in range(pop.numBays):
        MemNum = kk + BmCtr + 1
        writeBeamData(pop,story.beam[kk],output,MemNum)

def writeLatLoads(pop,building,output,loadFact,chkString):
    for jj in range(pop.numStories):
        if loadFact[3] < 0.0:
            JntNum = 1 + (jj + 1) * (pop.numBays + 1) + pop.numBays
            sign = -1.0
            # joint number for wind from right

```

```

else:
    JntNum = 1 + (jj + 1) * (pop.numBays + 1)
    sign = 1.0
if jj == (pop.numStories-1):          # roof level
    if chkString[0:1] == 's':         # wind only - no notional loads for service
        LatLoad = 0.50 * pop.typStryHt * pop.bayWidth * loadFact[3] * wind
    else:                              # wind & notional loads
        bmWt = 0                      # calc. wt. of beams for notional loads
        for m in range(pop.numBays):
            if m == 0 or m == pop.numBays-1:
                bmLength = pop.extBay
            else:
                bmLength = pop.intBay
            bmWt = bmWt + pop.dbBeams[building.story[jj].beam[m].shape]['plf'] * bmLength
        if pop.numBays <= 2:          # two bay frame with only exterior bays
            floorArea = pop.numBays * pop.extBay * pop.bayWidth
        else:
            numIntBays = pop.numBays - 2
            floorArea = (pop.intBay * numIntBays + 2 * pop.extBay) * pop.bayWidth
        LatLoad = loadFact[3]* wind * (pop.typStryHt / 2) * pop.bayWidth \
            + sign * ((loadFact[0] * roofDL + loadFact[2] * roofLL) * floorArea) * 0.002 \
            + sign * (bmWt) * 0.002
elif jj == 0:                        # first level
    if chkString[0:1] == 's':         # wind only - no notional loads for service level
        LatLoad = ((pop.firStryHt + pop.typStryHt) / 2) * pop.bayWidth * loadFact[3] * wind
    else:                              # wind & notional loads
        bmWt = 0                      # calc. wt. of beams for notional loads
        for m in range(pop.numBays):
            if m == 0 or m == pop.numBays-1:
                bmLength = pop.extBay
            else:
                bmLength = pop.intBay
            bmWt = bmWt + pop.dbBeams[building.story[jj].beam[m].shape]['plf'] * bmLength
        stryWt = 0                    # calc. wt. of story above for notional loads
        for j in range(1,pop.numStories):
            stryWt = stryWt + building.story[j].weight(pop)
        if pop.numBays <= 2:          # two bay frame with only exterior bays
            floorArea = pop.numBays * pop.extBay * pop.bayWidth
        else:
            numIntBays = pop.numBays - 2
            floorArea = (pop.intBay * numIntBays + 2 * pop.extBay) * pop.bayWidth

```

```

        LatLoad = loadFact[3] * wind * ((pop.firStryHt + pop.typStryHt) / 2) * pop.bayWidth \
            + sign * (loadFact[0] * roofDL + loadFact[2] * roofLL) * floorArea * 0.002 \
            + sign * (loadFact[0] * floorDL + loadFact[1] * floorLL) * (pop.numStories-jj-1) \
            * floorArea * 0.002 + sign * (bmWt + stryWt) * 0.002
else:
    # intermediate story level
    if chkString[0:1] == 's':
        # wind only - no notional loads for service level
        LatLoad = pop.typStryHt * pop.bayWidth * loadFact[3] * wind
    else:
        # wind & notional loads
        bmWt = 0
        # calc. wt. of beams for notional loads
        for m in range(pop.numBays):
            if m == 0 or m == pop.numBays-1:
                bmLength = pop.extBay
            else:
                bmLength = pop.intBay
            bmWt = bmWt + pop.dbBeams[building.story[jj]].beam[m].shape['plf'] * bmLength
        stryWt = 0
        # calc. wt. of story above for notional loads
        for j in range(jj+1,pop.numStories):
            stryWt = stryWt + building.story[j].weight(pop)
        if pop.numBays <= 2:
            # two bay frame with only exterior bays
            floorArea = pop.numBays * pop.extBay*pop.bayWidth
        else:
            numIntBays = pop.numBays - 2
            floorArea = (pop.intBay * numIntBays + 2 * pop.extBay) * pop.bayWidth
        LatLoad = loadFact[3] * wind * pop.typStryHt * pop.bayWidth \
            + sign * (loadFact[0] * roofDL + loadFact[2] * roofLL) * floorArea * 0.002 \
            + sign * (loadFact[0] * floorDL + loadFact[1] * floorLL) * (pop.numStories-jj-1) \
            * floorArea * 0.002 + sign * (bmWt + stryWt) * 0.002
OutString = `JntNum` + ' ' + `LatLoad` + ' ' + '0.00' + ' ' + '0.00'
output.write(OutString + '\n')

```

```

def writeGravLoads(pop,building,output,loadFact,chkString):
    for jj in range(pop.numStories-1):
        # nodal loads for column self wt.
        JntCtr = 1 + (jj+1) * (pop.numBays+1)
        for kk in range(pop.numBays+1):
            JntNum = JntCtr+kk
            col = building.story[jj+1].column[kk]
            ConcLd = pop.dbColumns[col.shape]['plf']*col.L
            OutString = `JntNum` + ' ' + '0.00' + ' ' + `ConcLd` + ' ' + '0.00'
            output.write(OutString + '\n')

```

```

for jj in range(pop.numStories):          # nodal loads for gravity loading & beam self wt.
    MemCtr = jj * (pop.numBays+1 + pop.numBays)
    for kk in range(pop.numBays):
        MemNum = kk + 1 + MemCtr + pop.numBays + 1
        if jj == pop.numStories-1:      # roof gravity loading
            UnifLd = (loadFact[0] * pop.bayWidth * roofDL \
                + loadFact[2] * pop.bayWidth * roofLL \
                + loadFact[0] * pop.dbBeams[building.story[jj].beam[kk].shape]['plf']) / 12.0
        else:                             # floor gravity loading
            UnifLd = (loadFact[0] * pop.bayWidth * floorDL \
                + loadFact[1] * pop.bayWidth * floorLL \
                + loadFact[0] * pop.dbBeams[building.story[jj].beam[kk].shape]['plf'])/12.0
        OutString = `MemNum` + ' ' + `UnifLd`
        output.write(OutString + '\n')

def writeBeamData(pop,beam,output,memb):
    OutString = `memb` + ' ' + `pop.dbBeams[beam.shape]['bf']` \
        + ' ' + `pop.dbBeams[beam.shape]['tf']` + ' ' + `pop.dbBeams[beam.shape]['d']` \
        + ' ' + `pop.dbBeams[beam.shape]['tw']` + ' ' + `pop.Fy` \
        + ' ' + `noSubBm`
    beam.number = memb
    output.write(OutString + '\n')
    writeConnectionData(pop,beam,beam.connection[0],output)
    writeConnectionData(pop,beam,beam.connection[1],output)

def writeConnectionData(pop,beam,connection,output):
    shape = beam.shape
    type = connection.type
    Theta1 = pop.dbConn[type][0][0] * 5 * pop.dbBeams[shape]['d'] * pop.dbBeams[shape]['Zx'] * pop.Fy \
        / (pop.E * pop.dbBeams[shape]['Ix'] )
    Theta2 = pop.dbConn[type][0][1] * 5 * pop.dbBeams[shape]['d'] * pop.dbBeams[shape]['Zx'] * pop.Fy \
        / (pop.E * pop.dbBeams[shape]['Ix'] )
    Theta3 = pop.dbConn[type][0][2] * 5 * pop.dbBeams[shape]['d'] * pop.dbBeams[shape]['Zx'] * pop.Fy \
        / (pop.E * pop.dbBeams[shape]['Ix'] )
    Stiff1 = pop.dbConn[type][1][0] * pop.dbBeams[shape]['Zx'] * pop.Fy / Theta1
    Stiff2 = (pop.dbConn[type][1][1] - pop.dbConn[type][1][0]) * pop.dbBeams[shape]['Zx'] * pop.Fy \
        / (Theta2 - Theta1)
    Stiff3 = (pop.dbConn[type][1][2] - pop.dbConn[type][1][1]) * pop.dbBeams[shape]['Zx'] * pop.Fy \
        / (Theta3 - Theta2)
    OutString = `Theta1` + ' ' + `Theta2` + ' ' + `Theta3` + ' ' \
        + `Stiff1` + ' ' + `Stiff2` + ' ' + `Stiff3`
    output.write(OutString + '\n')

```

```

def writeColumnData(pop,column,output,memb):
    OutString = `memb` + ' ' + `pop.dbColumns[column.shape]['bf']` \
        + ' ' + `pop.dbColumns[column.shape]['tf']` + ' ' + `pop.dbColumns[column.shape]['d']` \
        + ' ' + `pop.dbColumns[column.shape]['tw']` + ' ' + `pop.Fy` \
        + ' ' + `noSubCol`
    column.number = memb
    output.write(OutString + '\n')

def penalty(pop,limitInfo):
    penalties = []
    PHI_Gs = []
    PHI_DH = []
    PHI_DV = []
    PHI_ThS = []
    PHI_Eta = []
    PHI_ThU = []
    PHI_Gu = []
    PHI_Pn = []
    PHI_Kap = []
    PHI_Lpd = []
    PHI_htw = []
    PHI_shp = []
    for i in range(len(pop.building)):
        phi_Gs = 1.0
        phi_DH = 1.0
        phi_DV = 1.0
        phi_ThS = 1.0
        phi_Eta = 1.0
        phi_ThU = 1.0
        phi_Gu = 1.0
        phi_Pn = 1.0
        phi_Kap = 1.0
        phi_Lpd = 1.0
        phi_htw = 1.0
        phi_shp = 1.0

```

```

for j in range(pop.numStories-1):          # designer preference penalties
    for k in range(pop.numBays+1):
        lower = pop.building[i].story[j].column[k].shape
        upper = pop.building[i].story[j+1].column[k].shape
        if pop.dbColumns[upper]['dn'] > pop.dbColumns[lower]['dn']:
            phi_shp = calcPenalty([5.0,1],0.9,phi_shp)
        elif pop.dbColumns[upper]['dn'] == pop.dbColumns[lower]['dn']:
            if pop.dbColumns[upper]['plf'] > pop.dbColumns[lower]['plf']:
                phi_shp = calcPenalty([5.0,1],0.9,phi_shp)
            else:
                phi_shp = phi_shp
        else:
            if pop.dbColumns[upper]['plf'] > pop.dbColumns[lower]['plf']:
                phi_shp = calcPenalty([5.0,1],0.9,phi_shp)
            else:
                phi_shp = phi_shp
for ii in range(numLoadCases):
    fileName = 'evaluate/' + pop.building[i].outputFiles[ii]
    inputFile = open(fileName,'r')
    if ii <= 2 :                          # service load case
        Line1 = string.split(inputFile.readline())
        Gs     = string.atof(Line1[0])
        phi_Gs = calcPenalty([5.0,1],1.0/Gs,phi_Gs)
        for j in range(pop.numStories):    # beams
            for m in range(pop.numBays):
                Line = string.split(inputFile.readline())
                if m == 0 or m == pop.numBays-1:
                    DV_limit = (pop.extBay * 12.0) / limitInfo[1]

                else:
                    DV_limit = (pop.intBay * 12.0) / limitInfo[1]
                DV     = abs(string.atof(Line[4]))
                phi_DV = calcPenalty([5.0,1],DV/DV_limit,phi_DV)
                for n in range(2):          # connections
                    ThS      = abs(string.atof(Line[n+1]))
                    ThS_limit = connCurve(pop,pop.building[i].story[j].beam[m],ii,n)
                    phi_ThS  = calcPenalty([5.0,1],ThS/ThS_limit,phi_ThS)
                Yld = string.split(inputFile.readline())
                Yld.sort()
                Eta_limit = limitInfo[2]
                Eta      = abs(string.atof(Yld[len(Yld)-1]))
                phi_Eta  = calcPenalty([5.0,1],Eta/Eta_limit,phi_Eta)

```

```

for j in range(pop.numStories): # columns
    for k in range(pop.numBays+1):
        Line = string.split(inputFile.readline())
        DH_limit = (pop.building[i].story[j].column[k].L * 12.0) / limitInfo[0]
        DH = abs(string.atof(Line[2]))
        phi_DH = calcPenalty([5.0,1],DH/DH_limit,phi_DH)
        Yld = string.split(inputFile.readline())
        Yld.sort()
        Eta_limit = limitInfo[2]
        Eta = abs(string.atof(Yld[len(Yld)-1]))
        phi_Eta = calcPenalty([5.0,1],Eta/Eta_limit,phi_Eta)
    inputFile.close()
else: # ultimate load case
    Line1 = string.split(inputFile.readline())
    Gu = string.atof(Line1[0])
    phi_Gu = calcPenalty([5.0,1],1.0/Gu,phi_Gu)
    for j in range(pop.numStories): # beams
        for m in range(pop.numBays):
            Line = string.split(inputFile.readline())
            for n in range(2): # connections
                ThU = abs(string.atof(Line[n+1]))
                ThU_limit = connCurve(pop,pop.building[i].story[j].beam[m],ii,n)
                phi_ThU = calcPenalty([5.0,1],ThU/ThU_limit,phi_ThU)
            Kap_limit = limitInfo[3]*(pop.Fy/ \
                (pop.E*pop.dbBeams[pop.building[i].story[j].beam[m].shape['d']/2.0))
            Kap = abs(string.atof(Line[3]))
            phi_Kap = calcPenalty([5.0,1],Kap/Kap_limit,phi_Kap)
            skipLine = string.split(inputFile.readline())
    for j in range(pop.numStories): # columns
        for k in range(pop.numBays+1):
            Line = string.split(inputFile.readline())
            Pn = abs(string.atof(Line[3]))
            Pn_limit = weakAxisBuckling(pop,pop.building[i].story[j].column[k])
            phi_Pn = calcPenalty([5.0,1],Pn/Pn_limit,phi_Pn)
            Kap_limit = limitInfo[3]*(pop.Fy/ \
                (pop.E*pop.dbColumns[pop.building[i].story[j].column[k].shape['d']/2.0))
            Kap = abs(string.atof(Line[1]))
            phi_Kap = calcPenalty([5.0,1],Kap/Kap_limit,phi_Kap)
            Lpd = latTorBuckling(pop,pop.building[i].story[j].column[k],Line[4],Line[5])
            Lub = pop.building[i].story[j].column[k].L
            phi_Lpd = calcPenalty([5.0,1],Lub/Lpd,phi_Lpd)
            htw_limit = webLocalBuckling(pop,pop.building[i].story[j].column[k],string.atof(Line[3]))

```

```

                htw      = pop.dbColumns[pop.building[i].story[j].column[k].shape]['h/tw']
                phi_htw  = calcPenalty([5.0,1],htw/htw_limit,phi_htw)
                skipLine = string.split(inputFile.readline())
            inputFile.close()
        PHI_Gs.append(phi_Gs,i)
        PHI_ThS.append(phi_ThS,i)
        PHI_DH.append(phi_DH,i)
        PHI_DV.append(phi_DV,i)
        PHI_Eta.append(phi_Eta,i)
        PHI_Gu.append(phi_Gu,i)
        PHI_ThU.append(phi_ThU,i)
        PHI_Pn.append(phi_Pn,i)
        PHI_Kap.append(phi_Kap,i)
        PHI_Lpd.append(phi_Lpd,i)
        PHI_htw.append(phi_htw,i)
        PHI_shp.append(phi_shp,i)
    penalties.append(PHI_Gs)
    penalties.append(PHI_ThS)
    penalties.append(PHI_DH)
    penalties.append(PHI_DV)
    penalties.append(PHI_Eta)
    penalties.append(PHI_Gu)
    penalties.append(PHI_ThU)
    penalties.append(PHI_Pn)
    penalties.append(PHI_Kap)
    penalties.append(PHI_Lpd)
    penalties.append(PHI_htw)
    penalties.append(PHI_shp)

    return penalties

def calcPenalty(params,value,PhiValue):
    if value <= 1.0:
        PhiTmp = 1.0
    else:
        PhiTmp = 1.0 + params[0] * (value - 1.0) ** params[1]
    PhiValue = PhiValue * PhiTmp

    return PhiValue

```

```

def connCurve(pop,beam,loadCase,connNum):
    # Bjorhovde et. al., 1990
    Const = pop.dbBeams[beam.shape]['Zx'] * pop.Fy * 5.0 * \
            pop.dbBeams[beam.shape]['d'] / (pop.E * pop.dbBeams[beam.shape]['Ix'])
    type = beam.connection[connNum].type
    if loadCase <= 2:
        theta = pop.dbConn[type][0][0] * Const
    else:
        theta = (2.70 - 1.50 * pop.dbConn[type][1][2]) * Const

    return theta

def weakAxisBuckling(pop,column):
    # LRFD,1993 6-47
    slenParam = (column.L * 12.0 / (pop.dbColumns[column.shape]['ry'] * 3.14159265359)) * (pop.Fy / pop.E) ** 0.50
    if slenParam <= 1.50:
        Pny = 0.658 ** (slenParam ** 2) * pop.Fy * pop.dbColumns[column.shape]['A']
    else:
        Pny = (0.877 / slenParam ** 2) * pop.Fy * pop.dbColumns[column.shape]['A']

    return Pny

def latTorBuckling(pop,column,Ma,Mb):
    # LRFD,1993 6-55
    if abs(string.atof(Ma)) >= abs(string.atof(Mb)):
        M1 = string.atof(Mb)
        M2 = string.atof(Ma)
    else:
        M1 = string.atof(Ma)
        M2 = string.atof(Mb)
    Lpd = (3600.0 + 2200.0 * (M1 / M2)) * pop.dbColumns[column.shape]['ry'] / (pop.Fy / 1000.0) / 12.0

    return Lpd

def webLocalBuckling(pop,column,Pu):
    # LRFD,1993 6-175
    Py = pop.Fy * pop.dbColumns[column.shape]['A']
    if Pu / Py <= 0.125:
        htw = (640 / ((pop.Fy / 1000) ** (0.5))) * (1 - 2.75 * Pu / Py)
    else:
        htw = (191 / ((pop.Fy / 1000) ** (0.5))) * (2.33 - Pu / Py)
        if htw < 253 / ((pop.Fy / 1000) ** (0.5)):
            htw = 253 / ((pop.Fy / 1000) ** (0.5))

    return htw

```

```

#####
# MODULE: selection.py
#
# Module to SELECT frames based on individual fitness
#
# Open Source Python ver. 1.5
#####

import string
from Numeric import *
from RandomArray import *
import aisc
import copy

class roulette:
    def __init__(self,pop,PHI):
        self.newPop = []
        self.matingPool(pop)
        self.select(pop)
        saveBestIndiv(pop,PHI)
        pop.building = []
        for i in range(len(pop.fit)):
            pop.building.append(self.newPop[i])

    def matingPool(self,pop):
        fitSum = 0.0
        for i in range(len(pop.fit)):
            fitSum = fitSum + pop.fit[i][0]
        scaledFit = []
        for i in range(len(pop.fit)):
            scaledFit.append((fitSum/pop.fit[i][0],i))
        scaledFitSum = 0.0
        for i in range(len(pop.fit)):
            scaledFitSum = scaledFitSum + scaledFit[i][0]
        selectionProb = []
        for i in range(len(pop.fit)):
            selectionProb.append((scaledFit[i][0]/scaledFitSum,i))
        self.cumulFit = []
        cumulFitSum = 0.0
        for i in range(len(pop.fit)):
            cumulFitSum = cumulFitSum + selectionProb[i][0]
            self.cumulFit.append((cumulFitSum,i))

```

```

def select(self,pop):
    for i in range(len(pop.building)):
        ranNum = random()
        for j in range(len(pop.building)):
            if ranNum <= self.cumulFit[j][0]:
                self.newPop.append(copy.deepcopy(pop.building[j])) # all new memory addresses
                break

class tournament:
    def __init__(self,pop,PHI,tournSize,partBreak,probHighFit):
        self.size = tournSize
        self.partBreak = partBreak
        self.probHighFit = probHighFit
        self.newPop = []
        self.select(pop)
        saveBestIndiv(pop,PHI)
        pop.building = []
        for i in range(len(pop.fit)):
            pop.building.append(self.newPop[i])

    def select(self,pop):
        pop.fit.sort()
        partIndx = int(self.partBreak*len(pop.building))
        for i in range(len(pop.building)):
            tournGrp = []
            for j in range(self.size):
                if random() <= self.probHighFit: # select from the "better partition"
                    theOne = randint(0,partIndx)
                else: # select from the "rest"
                    theOne = randint(partIndx,len(pop.building)-1)
                tournGrp.append((pop.fit[theOne][0],pop.fit[theOne][1]))
            tournGrp.sort()
            winner = tournGrp[0][1]
            self.newPop.append(copy.deepcopy(pop.building[winner])) # all new memory addresses

```

```
def saveBestIndiv(pop,PHI):
    fitFeas = []
    for i in range(len(pop.building)):
        compProduct = 1.0
        for j in range(len(PHI)):
            compProduct = compProduct * PHI[j][i][0]
        if compProduct <= 1.0:
            fitFeas.append(pop.wt[i][0],i)
    if len(fitFeas) == 0:
        pop.fit.sort()
        bestNum = pop.fit[0][1]
    else:
        fitFeas.sort()
        bestNum = fitFeas[0][1]
    pop.bestIndiv = pop.building[bestNum]
```

```

#####
# MODULE: reproduction.py
#
# Module to CROSSOVER and MUTATE members of frames
#
# Open Source Python ver. 1.5
#####

import string
from Numeric import *
from RandomArray import *
import aisc
import copy

class crossover:
    def __init__(self, pop, colRate, bmRate, conRate):
        self.nextGen = []
        self.colRate = colRate
        self.bmRate = bmRate
        self.conRate = conRate
        if pop.varType == 'indiv':
            self.indivXover(pop)
        else:
            self.grpXover(pop)
        pop.building = []
        for i in range(len(self.nextGen)):
            pop.building.append(self.nextGen[i])

    def indivXover(self, pop):
        # only swapping attributes (i.e. shape & type)
        for i in range(len(pop.building)):
            # select control individual (loop over the entire population)
            ii = self.getMate(i, len(pop.building))
            # select mate (randomly from rest of population)
            A = copy.deepcopy(pop.building[i])
            # offspring - all new memory addresses
            for j in range(pop.numStories):
                for k in range(pop.numBays+1):
                    if random() <= self.colRate:
                        if random() < 0.5:
                            # homologous
                            A.story[j].column[k].shape = pop.building[ii].story[j].column[k].shape
                        else:
                            # nonhomologous
                            jj = self.getMate(j, pop.numStories)
                            kk = randint(0, pop.numBays+1)
                            A.story[j].column[k].shape = pop.building[ii].story[jj].column[kk].shape

```

```

for m in range(pop.numBays):
    if random() <= self.bmRate:
        if random() < 0.5:          # homologous
            A.story[j].beam[m].shape = pop.building[ii].story[j].beam[m].shape
        else:                      # nonhomologous
            jj = self.getMate(j, pop.numStories)
            mm = randint(0, pop.numBays)
            A.story[j].beam[m].shape = pop.building[ii].story[jj].beam[mm].shape
    if random() <= self.conRate:
        if random() < 0.5:        # homologous
            for n in range(2):
                A.story[j].beam[m].connection[n].type = \
                    pop.building[ii].story[j].beam[m].connection[n].type
        else:                      # nonhomologous
            jj = self.getMate(j, pop.numStories)
            mm = randint(0, pop.numBays)
            for n in range(2):
                A.story[j].beam[m].connection[n].type = \
                    pop.building[ii].story[jj].beam[mm].connection[n].type

self.nextGen.append(A)

def grpXover(self, pop):
    for i in range(len(pop.building)):          # only swapping attributes (i.e. shape & type)
        ii = self.getMate(i, len(pop.building)) # select control individual (loop over the entire population)
        A = copy.deepcopy(pop.building[ii])     # select mate (randomly from rest of population)
        for j in range(pop.numStories):        # offspring - all new memory addresses
            if random() <= self.colRate:       # exterior columns
                if random() < 0.5:             # homologous
                    A.story[j].column[0].shape = pop.building[ii].story[j].column[0].shape
                else:                          # nonhomologous
                    jj = self.getMate(j, pop.numStories)
                    A.story[j].column[0].shape = pop.building[ii].story[jj].column[0].shape
            if random() <= self.colRate:       # interior columns
                if random() < 0.5:             # homologous
                    A.story[j].column[1].shape = pop.building[ii].story[j].column[1].shape
                else:                          # nonhomologous
                    jj = self.getMate(j, pop.numStories)
                    A.story[j].column[1].shape = pop.building[ii].story[jj].column[1].shape
            if random() <= self.bmRate:
                if random() < 0.5:             # homologous
                    A.story[j].beam[0].shape = pop.building[ii].story[j].beam[0].shape
                else:                          # nonhomologous

```

```

        jj = self.getMate(j,pop.numStories)
        A.story[j].beam[0].shape = pop.building[ii].story[jj].beam[0].shape
    if random() <= self.conRate:
        if random() < 0.5:                # homologous
            A.story[j].beam[0].connection[0].type = \
            pop.building[ii].story[j].beam[0].connection[0].type
        else:                             # nonhomologous
            jj = self.getMate(j,pop.numStories)
            A.story[j].beam[0].connection[0].type = \
            pop.building[ii].story[jj].beam[0].connection[0].type
    self.nextGen.append(A)

```

Note: No looping occurs at the column, beam and connection level in grpXover() since the 'grp' variables have the same memory address. For example, changing (crossing over) beam[0].shape will change all beams for that story.

```

def getMate(self,counter,limit):          # selects mate that is not the same control individual
    mate = randint(0,limit)
    if counter == mate:
        if counter == limit-1:
            mate = mate - 1
        else:
            mate = mate + 1
    return mate

```

```

class mutate:
    def __init__(self,pop,bldgRate,stryRate,colRate,bmRate,conRate):
        self.bldgRate = bldgRate
        self.stryRate = stryRate
        self.colRate = colRate
        self.bmRate = bmRate
        self.conRate = conRate
        self.building(pop)
        self.story(pop)
        self.column(pop)
        self.beam(pop)
        self.connection(pop)

```

Note: Since the buildings are instantiated according to 'indiv' and 'grp' variables, the memory addresses for each object have been instantiated accordingly. Therefore, there is no need for separate 'indiv' and 'grp' mutation methods for column, beam, and connection mutation. For example, if the ext. columns are mutated for 'grp' variables, then both ...column[0] & ...column[pop.numBays] are mutated with a single call to the setSize method: pop.building[i].story[j].column[0].setSize()

```

def building(self,pop):
    for i in range(len(pop.building)):
        if random() <= self.bldgRate:
            pop.building[i].story = []
            pop.building[i].setStories(pop)

def story(self,pop):
    for i in range(len(pop.building)):
        if random() <= self.stryRate:
            storyLoc = randint(0,pop.numStories-1)
            pop.building[i].story[storyLoc].beam = []
            pop.building[i].story[storyLoc].column = []
            if pop.varType == 'indiv':
                pop.building[i].story[storyLoc].setIndivSizes(pop)
            else:
                pop.building[i].story[storyLoc].setGrpSizes(pop)

def column(self,pop):
    for i in range(len(pop.building)):
        if random() <= self.colRate:
            storyLoc = randint(0,pop.numStories-1)
            bayLoc = randint(0,pop.numBays)
            pop.building[i].story[storyLoc].column[bayLoc].setSize(pop)

def beam(self,pop):
    for i in range(len(pop.building)):
        if random() <= self.bmRate:
            storyLoc = randint(0,pop.numStories-1)
            bayLoc = randint(0,pop.numBays-1)
            pop.building[i].story[storyLoc].beam[bayLoc].setSize(pop)

def connection(self,pop):
    for i in range(len(pop.building)):
        if random() <= self.conRate:
            storyLoc = randint(0,pop.numStories-1)
            bayLoc = randint(0,pop.numBays-1)
            conLoc = randint(0,2)
            pop.building[i].story[storyLoc].beam[bayLoc].connection[conLoc].setType(pop)

def elitism(pop):
    A = randint(0,len(pop.building))
    pop.building[A] = pop.bestIndiv

```

```

#####
# MODULE: aisc.py
#
# Module to create and access AISC W-shape properties for beam and column members
#
# Open Source Python
#####

import string

def readWshapes(type):
    if type == 'beam':
        numWshapes = 149
        input = open('aiscbeam.dat','r')
    else:
        numWshapes = 71
        input = open('aisccol.dat','r')
    wdata = []
    for i in range(numWshapes):
        wdata.append(input.readline())
    input.close()
    db = {}
    for shape in wdata:
        tempdict={}
        tempdict[ 'dn' ] = string.atof( shape[ 4-1: 8] ) # 'DN'
        tempdict[ 'plf' ] = string.atof( shape[ 9-1: 14] ) # 'WGT'
        tempdict[ 'jumbo' ] = shape[ 15-1: 16] # 'JSHP'
        tempdict[ 'A' ] = string.atof( shape[ 17-1: 23] ) # 'A'
        tempdict[ 'd' ] = string.atof( shape[ 24-1: 30] ) # 'D'
        tempdict[ 'tw' ] = string.atof( shape[ 31-1: 37] ) # 'TW'
        tempdict[ 'bf' ] = string.atof( shape[ 38-1: 44] ) # 'BF'
        tempdict[ 'tf' ] = string.atof( shape[ 45-1: 50] ) # 'TF'
        tempdict[ 'k' ] = string.atof( shape[ 51-1: 56] ) # 'XK'
        tempdict[ 'bf/2tf' ] = string.atof( shape[ 57-1: 60] ) # 'BTF'
        tempdict[ 'fyp' ] = string.atof( shape[ 61-1: 64] ) # 'FYP'
        tempdict[ 'h/tw' ] = string.atof( shape[ 65-1: 68] ) # 'HTW'
        tempdict[ 'd/tw' ] = string.atof( shape[ 69-1: 72] ) # 'DTW'
        tempdict[ 'fypppL' ] = string.atof( shape[ 73-1: 76] ) # 'FYPPPL'
        tempdict[ 'fypppA' ] = string.atof( shape[ 77-1: 80] ) # 'FYPPPA'
        tempdict[ 'x1' ] = string.atof( shape[ 81-1: 85] ) # 'X1'
        tempdict[ 'x2' ] = string.atof( shape[ 86-1: 93] ) # 'X2'
        tempdict[ 'rt' ] = string.atof( shape[ 94-1: 98] ) # 'RT'

```

```

tempdict[ 'd/AF' ] = string.atof( shape[ 99-1:104 ] ) # 'DAF'
tempdict[ 'RI' ] = string.atof( shape[105-1:109] ) # 'RI'
tempdict[ 'RA' ] = string.atof( shape[110-1:114] ) # 'RA'
tempdict[ 'NT' ] = string.atof( shape[115-1:119] ) # 'NT'
tempdict[ 'Ix' ] = string.atof( shape[116-1:124] ) # 'XI'
tempdict[ 'Sx' ] = string.atof( shape[125-1:132] ) # 'SX'
tempdict[ 'rx' ] = string.atof( shape[133-1:138] ) # 'RX'
tempdict[ 'Iy' ] = string.atof( shape[139-1:147] ) # 'YI'
tempdict[ 'Sy' ] = string.atof( shape[148-1:154] ) # 'SY'
tempdict[ 'ry' ] = string.atof( shape[155-1:160] ) # 'RY'
tempdict[ 'Zx' ] = string.atof( shape[161-1:168] ) # 'ZX'
tempdict[ 'Zy' ] = string.atof( shape[169-1:176] ) # 'ZY'
tempdict[ 'J' ] = string.atof( shape[177-1:186] ) # 'XJ'
tempdict[ 'Cw' ] = string.atof( shape[187-1:196] ) # 'CW'
tempdict[ 'Wno' ] = string.atof( shape[197-1:204] ) # 'WNO'
tempdict[ 'Sw' ] = string.atof( shape[205-1:212] ) # 'SW'
tempdict[ 'Qf' ] = string.atof( shape[213-1:220] ) # 'QF'
tempdict[ 'Qw' ] = string.atof( shape[221-1:228] ) # 'QW'
tempdict[ 'ro' ] = string.atof( shape[229-1:234] ) # 'RO'
tempdict[ 'H' ] = string.atof( shape[235-1:240] ) # 'H'

aiscshape = string.strip(shape[1-1: 3])
aiscdepth = string.strip(shape[4-1: 8]) # work around required for aisc W4, W5, W6, W8,
aiscdepth = aiscdepth[0:len(aiscdepth)-3] # format error in AISC file
aiscweight = string.strip(shape[9-1:14-3])
designation = string.join((aiscshape,aiscdepth,'X',aiscweight),'')
db[designation] = tempdict

dbkeys = db.keys()
dbkeys.sort()

return db

```

```

def makeBmList(db,key1,key2,key3):                # make a beam list table with three keys
    BmList = []
    for key in db.keys():
        BmList.append(db[key]['plf'],db[key][key1],db[key][key2],db[key][key3],key)
    BmList.sort()

    return BmList

def makeColList(db,key1,key2):                   # make a column list table with four keys
    ColList = []
    for key in db.keys():
        ColList.append(db[key]['plf'],db[key][key1],db[key][key2],key)
    ColList.sort()

    return ColList

def getBmList(BmList,var1,var2,var3):           # Establish beam series satisfying var1 - var3
    Beams = []
    for item in BmList:
        if item[1] == var1 and item[2] <= var2 and item[3] <= var3:
            Beams.append(item)

    return Beams

def getColList(ColList,var1,var2):              # Establish column series satisfying var1 - var4
    Columns = []
    for item in ColList:
        if item[1] == var1 and item[2] <= var2:
            Columns.append(item)

    return Columns

```

```

#####
# MODULE: cdata.py
#
# Module to create and access a database of connection stiffness and weight modification factors
#
# Open Source Python
#####

import string

def readConnData(numConnections):
    db = {}
    input = open('conndata.dat', 'r')          # access connection database (based on Bjorhovde Model)
    for i in range(numConnections):
        line = string.split(input.readline())
        if line[0] == 'SPR':
            list1 = []                          # list of non-dimensional rotations
            list2 = []                          # list of non-dimensional moment capacities
            list3 = []                          # list of weight factors
            key = 'C'+`i+1`
            for j in range(3):
                list1.append(string.atof(line[j+1]))
            for j in range(3):
                list2.append(string.atof(line[j+4]))
            for j in range(3):
                list3.append(string.atof(line[j+7]))
            db[key] = [list1,list2,list3]

    input.close()

    return db

```

Appendix C
Enumeration Study (Chapter 5)

This appendix provides the calculations to enumerate all combinations of member shapes and connection types for the three frames designed in Chapter 5. Both fully-restrained (FR) and partially-restrained (PR) connections are considered and the design variables are grouped as defined in Chapter 4. The following equation is used to determine the *Total* number of combinations for frames:

$$Total = [p(n_{col}, r_{col}) + n_{col}] \cdot [p(n_{bm} \cdot n_{conn}, r_{bm}) + n_{bm} \cdot n_{conn}]$$

where,

$$p(n_{col}, r_{col}) = \frac{n_{col}!}{(n_{col} - r_{col})!} \text{ is the number of column permutations}$$

n_{col} = number of available columns (equal to 71 for all frames)

r_{col} = number of columns within a frame (dependent on frame configuration)

$$p(n_{bm} \cdot n_{conn}, r_{bm}) = \frac{(n_{bm} \cdot n_{con})!}{(n_{bm} \cdot n_{con} - r_{bm})!} \text{ is the number of beam \& connection permutations,}$$

n_{bm} = number of available beams (equal to 149 for all frames)

r_{bm} = number of beams within a frame (dependent on frame configuration)

n_{con} = number of available connections (equal to 5 for partially-restrained frames and one for fully-restrained frames)

Frame 1 (three stories, two bays):

FR connections:

$$Total = [p(71, 6) + 71] \cdot [p(149 \cdot 1, 3) + 149 \cdot 1] = 3.343 \cdot 10^{17}$$

PR connections:

$$Total = [p(71, 6) + 71] \cdot [p(149 \cdot 5, 3) + 149 \cdot 5] = 4.247 \cdot 10^{19}$$

Frame 2 (two stories, three bays):***FR connections:***

$$Total = [p(71,4) + 71] \cdot [p(149 \cdot 1, 2) + 149 \cdot 1] = 5.177 \cdot 10^{11}$$

PR connections:

$$Total = [p(71,4) + 71] \cdot [p(149 \cdot 5, 2) + 149 \cdot 5] = 1.294 \cdot 10^{13}$$

Frame 3 (ten stories, 3 bays):***FR connections:***

$$Total = [p(71,20) + 71] \cdot [p(149 \cdot 1, 10) + 149 \cdot 1] = 2.172 \cdot 10^{57}$$

PR connections:

$$Total = [p(71,20) + 71] \cdot [p(149 \cdot 5, 10) + 149 \cdot 5] = 2.718 \cdot 10^{64}$$

The required number of combinations calculated above illustrates the impracticality of enumerating the entire solution space for the design frame examples in Chapter 5.