Shear wall layout optimization for conceptual design of tall buildings

by

Yu Zhang

B.Eng. in Civil Engineering, Southwest Jiaotong University (2014)

Submitted to the Department of Civil and Environmental Engineering in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author .... Signature redacted

Department of Civil and Environmental Engineering

May 12, 2017

Certified by .... Signature redacted

Caitlin Mueller

Assistant Professor of Architecture and Civil and Environmental Engineering

Thesis Supervisor

Accepted by .... Signature redacted ....

Jesse Kroll

Professor of Civil and Environmental Engineering

Chair, Graduate Program Committee

Shear wall layout optimization for conceptual design of tall buildings

by

Yu Zhang

Submitted to the Department of Civil and Environmental Engineering
on May 12, 2017, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

In the design of tall buildings, the lateral system that resists wind and seismic loading usually dominates the structural engineering effort; therefore, optimal lateral system design is important for material efficiency. In a shear-wall-based building, the conventional design process starts with an architect generating a floor plan, which is then passed to a structural engineer, who, based on knowledge and prior experience, tries to place shear walls to balance conflicting requirements: minimum structural weight, satisfactory structural strength and serviceability, conformity to architectural layout. This design process can be slow and inefficient, requiring a trial-and-error approach that is unlikely to lead to the best solution. The work presented in this thesis intends to accelerate the process with an optimization system involving a ground structure program formulation, a modified evolutionary algorithm, and innovative computational techniques. Unlike existing work that focuses either exclusively on structural performance or architectural layout, this research integrates both. An efficient computational design methodology for shear wall layout in plan is introduced. The method minimizes structural weight with constraints on torsion, flexural strength, shear strength, drift, and openings and accessibility. It can be applied from the very beginning of floor plan design or after generating an architectural floor plan. This thesis demonstrates the potential of this approach through a variety of case studies. Key contributions include a novel application of the ground structure method, a fast and robust modified evolutionary algorithm, and a simplified auto-calculation system for reinforced concrete design.

Thesis Supervisor: Caitlin Mueller
Title: Assistant Professor of Architecture and Civil and Environmental Engineering

To Professor Caitlin Mueller

Acknowledgments

I would first like to thank my thesis advisor Professor Caitlin Mueller, who has consistently provided advice on my research as well as my life decisions. I was introduced to the world of structral optimization by Prof. Mueller's class and I am grateful that Professor Mueller provided me with this opportunity to work on structural optimization for my thesis. Professor Mueller consistently allowed this thesis to be my own work, but steered me in the right the direction and offered significantly useful advice whenever she thought I needed it. I would also like to thank Ramon Gilsanz from Gilsanz Murray Steficek, who originally and insightfully proposed this research topic in July 2015.

Besides, I would like to thank the Department of Civil and Environmental Engineering at M.I.T., especially Professor John Ochsendorf, for the valuable comments on this thesis and support on my research. The M.Eng./S.M program offered me the chance to come to this prestigious institute, where my horizion is broadened, my research skill is improved, and my life is delighted with so many amazing people.

I would also like to thank all the staff involved in MIT-SUTD Fellowship Program, especially Mr. Jonathan Griffith and Professor James Wan, who provided me with the fellowship offer that enabled me to continue my research and to gain the invaluable experience as an teaching assistant in Singapore University of Technology and Design. This thesis would not have been accomplished without the financial support from the MIT-SUTD Fellpwship Program.

Finally, I must express my very profound gratitude to my parents and to my boyfriend Zhao Ma for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the design of tall buildings, the lateral system that resists wind and earthquake loads often dominates. Reinforced concrete shear walls (example shown in Figure 1-1), a common type of the lateral system often arranged around elevators and other vertically continuous building elements as shear cores, require special consideration both structurally and architecturally because of their impact on spatial arrangement in plan. The requirements from architects, engineers and clients are usually conflicting and highly interconnected: for example, architects may focus on the spatial interaction and arrangement, natural lighting, and accessibility; structural engineers would like to place a sufficient amount of shear walls to satisfy the structural strength and serviceability; clients, however, would prefer the cost of material and labor to be as low as possible. A highly symmetric floor plan can address this issue, appearing organized to the architect and efficient to the structural engineer. However, for buildings with irregularity (irregular shape in plan, or with atriums or spacious wall-free area), which are very common contemporarily, the process becomes considerably more complex and interdisciplinary. Traditionally, to design a floor plan with an asymmetrical layout, architects and structural engineers, working separately, rely on a wide range of inputs, including intuition, experience, rules of thumb, analytical modeling, and simulation, all combined in an iterative design-and-test process. Although a valid layout might be obtained consequently, there is no way to guarantee it is optimized in terms of any particular goal.

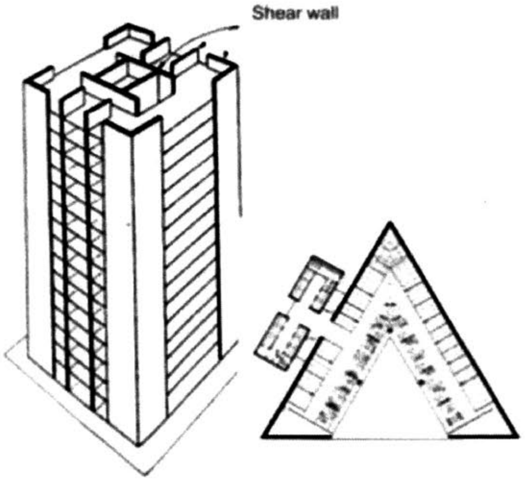

Figure 1-1: Example of shear walls in a high-rise buidling. (Shivank Sharma, 2016)

1.2 Related work

Currently, there is limited research directly on this topic. Most existing literature focuses on related topics either from an exclusively structural or architectural perspective. In the structural engineering field, most research looks at the optimization of lateral systems in 2D elevation view, rather than the layout in plan. For example, Chan and Wong used hybrid OC-GA method to generate both sizes and topologies at the same time for braced frame structures (Figure 1-2) [3]. Liang et al. used performance-based design method to generate optimal structural design by removing inefficient materials from a continuum design domain (Figure 1-3) [11]. While this type of research is important and useful, it does not address the organization of the lateral system in plan and its inherent interactions with architectural decision-making.

In the architectural field, floor plan layout topology has been studied: Peng et al. proposed an integer programming-based approach to model corridors in a floor plan, where they defined a set of room templates and generated a subset of all possible po-

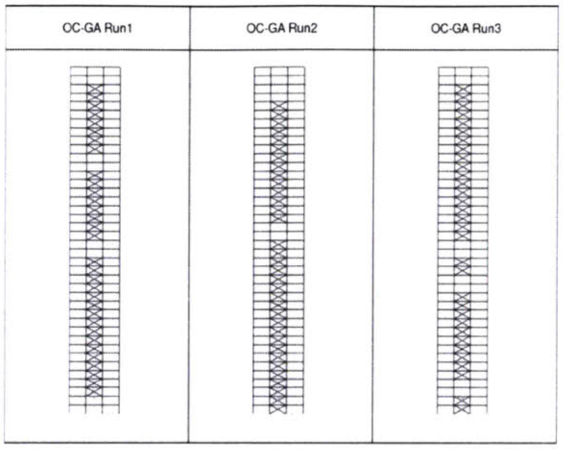

Figure 1-2: Hybrid OC-GA method. (Chan and Wong, 2008)

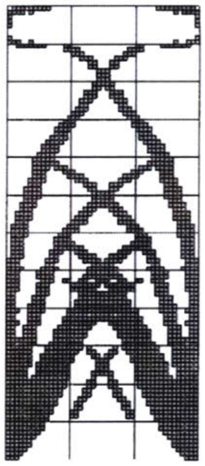

Figure 1-3: Using performance-based design method to generate optimal structural design by removing inefficient materials from a continuum design domain. (Liang et al., 2000)

tential room placements such that no two overlap and together cover the area (Figure 1-4) [14]. Lai used graph theory for floor plan design where the rectangular dualization problem was reduced to a matching problem on bipartite graphs [7]. Terzidis developed software called AutoPLAN that generates architectural floor plans with the boundary and adjacency matrix of a given site (Figure 1-5) [15]. Stiny proposed the formulation of floor plans using a shape grammar, which is a rulebased procedure for generating different geometric shapes [6]. Shekhawat developed an algorithm which takes connectivity into consideration (Figure 1-6) [10]. Nevertheless, these floor plan optimizations are more relevant to architectural design than structural engineering. The lateral system is not adequately considered, which would affect the structural performance of high-rise buildings and lead to laborious iterations due to the separation between architectural and structural approaches during the design process.

Aminnia et al. looked for the optimal patterns by locating the components of shear wall lateral systems, but the configuration of shear walls was constrained to be T-shaped, Z-shaped, U-shaped or L-shaped beforehand [2]. The broader general topology optimization problem remains unaddressed and is thus one of the main focuses of this thesis.

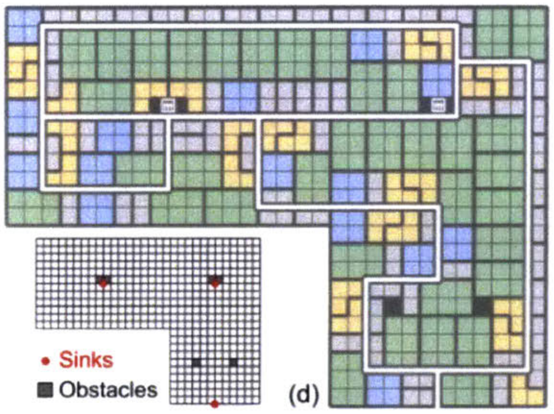

Figure 1-4: An integer programming-based approach to model corridors in a floor plan. (Peng et al., 2016)

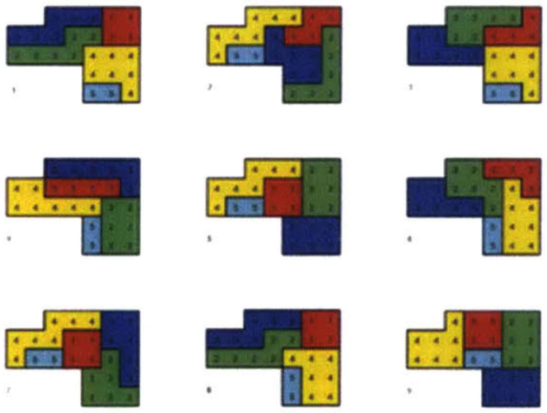

Figure 1-5: AutoPLAN. (Terzidis, 2007-2008)

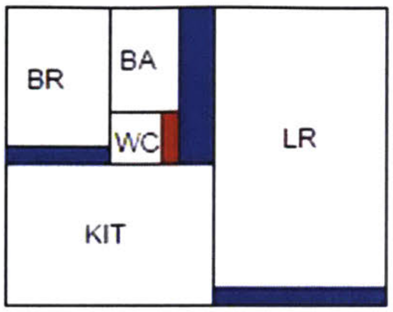

Figure 1-6: Algorithm that takes connectivity into consideration. (Shekhawat, 2014)

1.3 Organization of thesis

The research presented in this thesis aims to develop a computational method for producing architecture-compatible shear wall layouts with minimized structural weight under basic structural analysis. A simple example of the desired result is shown in Figure 1-7. Chapter 2 gives a detailed description of the general methodology: Section 2.2.2 introduces a novel ground structure method which discretizes a given building footprint into a quadrilateral mesh with each edge modeled as a potential location of a shear wall member. By activating and deactivating shear walls on these edges of the mesh, different possible layouts can be generated. Section 2.2.3 proposes a simplified method for reinforced concrete design regarding irregularly configured shear walls. Due to the large number of possible layouts and a complex objective function containing both linear and nonlinear constraints, a modified evolutionary algorithm is introduced in Section 2.3. Since the optimal set of optimization parameters varies with different buildings, effectively selecting parameters is essential and the method is described in Section 2.4. Chapter 3 illustrates the results and evaluation of a simple case study. In Chapter 4, this thesis introduces more complex case studies with different combinations of irregular contour and void spaces, or with fixed floor plans. This illustrates applications in two design phases: (1) during the floor plan design phase, which brings more flexibility and inspiration to architects and simultaneously allows more usability (Section 4.1), and (2) after floor plan design phase, when architects have already arranged the space and thus yielded a fixed floor plan to the structural engineers who would like to know the optimal shear wall layout for this specific case (Section 4.2). Finally, Chapter 5 summarizes the contributions of the thesis and suggests areas for further research.

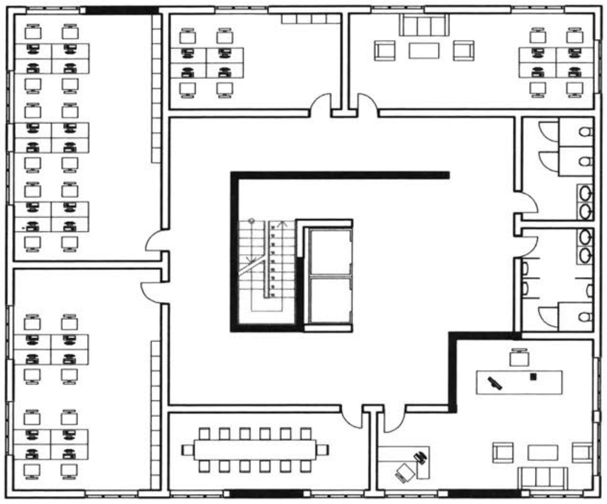

Figure 1-7: Example of a desired final layout result. The solid lines represent locations chosen for the shear walls.

Chapter 2

General methodology

2.1 Conceptual overview

This research develops a system to optimize the layout of shear walls in terms of structural weight, under structural and architectural constraints. To illustrate the methodology simply and clearly, this thesis assumes that the default input problem domain is a 2D rectangular building footprint (more variations are discussed in Chapter 4). The footprint is firstly discretized into a quadrilateral mesh with each edge representing a potential shear wall location. This mesh is modeled as an unconventional ground structure, such as those seen in the topology optimization of truss structures [4], allowing the shear wall on each edge to be either activated or deactivated. For the purpose of optimization, a modified evolutionary algorithm is introduced. The goal is to minimize structural weight with penalties on structural requirements (torsion, flexural, shear, and drift) and on basic architectural requirements (accessibility and openings) to be detailed in later chapters. With algorithmic parameters defined by users, this evolutionary algorithm mimics biological evolution by iterating through several cycles of reproduction and selection. However, to accelerate the evolving speed, biased and directional pairing and mutation is applied in the reproduction process. The subset of best-performing individual layouts in the last generation is considered the result of this optimization.

In this thesis, a twenty-story residential building in Boston, with dimensions of 80ft $\times$ 60ft $\times$ 240ft tall (24m $\times$ 18m $\times$ 70m tall), is set as a simple example to illustrate the method. With each shear wall element being 10ft (3m) long, the footprint of this building is meshed into a 6 by 8 grid.

2.2 Setup

2.2.1 Parameters

Defined at the model initialization step and remaining unchanged during optimization process, a parameter set consists of algorithmic parameters and structural parameters. Algorithmic parameters include evolutionary parameters determining the optimization speed and diversity (such as mutation rate, generation size, and number of generations), and objective function parameters adjusting the evaluation standard. While algorithmic parameters are subjective and affect the general optimization capacity of the model, structural parameters (including building geometry parameters, shear wall property parameters, and loading and deflection parameters) are more objective and usually vary with different cases. Thus by making the parameters modifiable for users, a custom-built system can be obtained depending on specific needs. A simplified method for parameter selection is demonstrated in Section 2.4.

In a specific site, given the purpose, dimensions (building geometry parameters), and material of the building, the loading can be estimated (dead load, live load, wind load, etc.). Under strength or ultimate limit state (ULS) design, buildings shall be designed to resist the most critical effects resulting from various combinations of factored loads. While serviceability design has several common parameters including deflection, vibration, slenderness, and clearance, this thesis only takes drift constraints into consideration which are determined and implemented in the objective function of the evolutionary algorithm.

In the twenty-story building example, information of loading and load combinations considered under ULS design and serviceability design [1] is presented as follows: ULS: 1.2D + 1.6W + 1.0L; Serviceability: 1.0D + 1.0W + 1.0L, where D, W, and L are dead load, wind load, and live load respectively. For loading values, see Table 2.1.

Table 2.1: Loading values used

Dead Load (D)	Live Load (L)	Wind Load (W)
170psf (8.14kPa)	45psf (2.15kPa)	30psf (1.44kPa)

2.2.2 Ground structure

The ground structure method (Figure 2-1), developed by Dorn et al., achieves the optimal result by optimizing over a wide class of possible structures whose elements are all selected from a prescribed admissible set [4]. This method is widely applied in topology optimization, especially for trusses with finite nodes and bars. Hagishita and Ohsaki (2009) improved the ground structure method by proposing adding and removing bars and nodes based on some growing rules, which is called the growing ground structure method [8]. Sokó et al. modified the adaptive ground structure approach, which can be used for multi-load truss topology optimization [5].

While the ground structure method mentioned above usually considers elements connecting any two nodes, most of the shear wall layout domain can be covered by an orthogonal grid, which is simple and efficient in terms of computational cost. Thus this thesis utilizes a ground structure generated by discretizing a building footprint into a quadrilateral mesh with each edge representing a potential reinforced concrete shear wall. All of these edges are sequenced with indices (Figure 2-2a) and each possesses state as either activated or deactivated, indicating the presence or absence of a shear wall respectively (Figure 2-2c). A shear wall layout is a subset of the edges

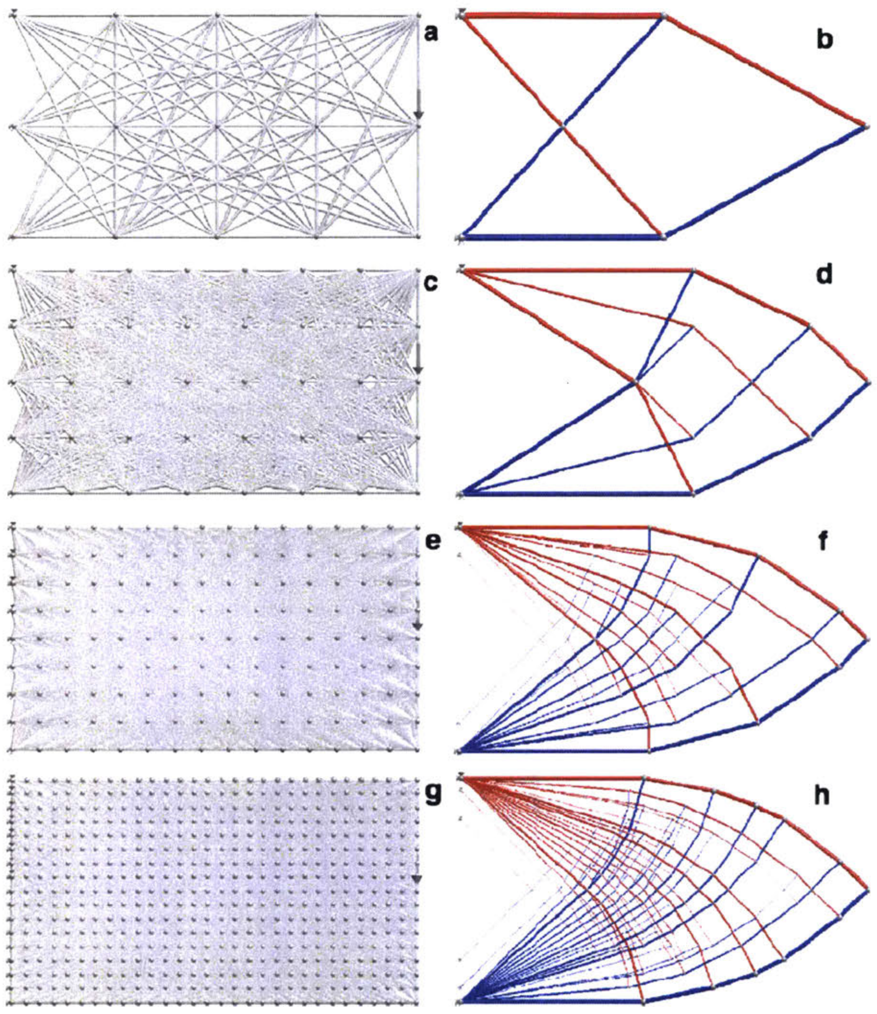

Figure 2-1: Example of ground structure (P. Martínez, 2007).

in this quad mesh. Though these features can be achieved using object-oriented programming, this representation method lacks efficiency in the forthcoming evolution stage. For the convenience of computational modeling, the subset is recorded by a binary string ($\boldsymbol{v}$), resembling a chromosome. Like the function of chromosome, this binary string carries and conveys topology information. Each digit of the binary number represents an edge in the mesh and the number on each digit indicates status of this specific edge ("0" indicates deactivated and 1 indicates activated, see Figure 2-2b).Thus every possible layout can be represented by a unique chromosome (example shown in Figure 2-2b).

For unconventional floorplans where shear walls need to be placed in diagonal di-

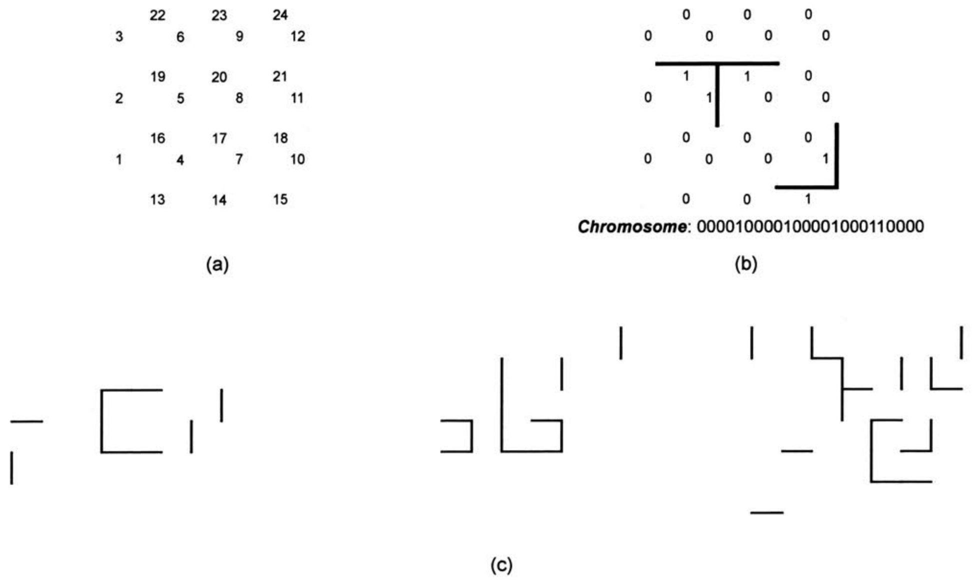

Figure 2-2: Examples of the ground structure for a building footprint: (a) shows every edge has a unique index numbered by its sequence, (b) illustrates that the status of each edge is indicated by either "0" or "1", and (c) shows three examples of 6 by 8 grids with activated and deactivated walls.

rections, users can predefine the design domain to allow diagonal lines and extend the binary string ($\boldsymbol{v}$) accordingly. For simplicity, this thesis will not present this version in detail in the following content.

2.2.3 Reinforced concrete analysis

The method presented in this thesis considers the strength and stiffness of reinforced concrete shear wall layouts with a simplified design method that includes assumptions related to cracking. While additional objectives, including creep and shrinkage, should be considered later in the detailing stage of the design process, these two criteria are considered most critical for conceptual design. The parameters of wall dimensions, reinforcement characteristics, and material properties should be decided before optimization, where they are involved in a check against strength and drift constraints, which are the main considerations for structural design in this thesis. For the twenty-story building case, the selected values are shown in Table 2.2. For programming implementation and economic reasons, reinforcement in this thesis is assumed to possess uniform cross-section and distribution. However, this might be subject to change or manual check if the building is in moderate or high seismic zones. In that case, concentration of reinforcement at the extreme ends of a wall is usually helpful [13].

Table 2.2: Parameters for the reinforced concrete by default.

	Strength	Dimension
Concrete	5ksi (35MPa)	10ft $\times$ 1ft (305cm $\times$ 30.5cm)
Vertical Steel	60ksi (414MPa)	#8 (2.54cm dia.) 10in (25.4cm) $\times$ 2 layers

Since the ground structure method may produce some layouts with connected shear wall groups, as shown in Figure 2-3, it is important to analyze each groups structural performance and contribution to the overall behavior and performance of the lateral system. The research presented in this thesis assumes that lateral loads are distributed to each shear wall group, according to its relative rigidity, by in-plane action of the rigid floor plate diaphragm.

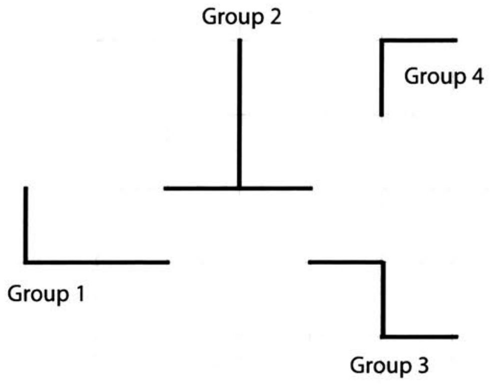

Figure 2-3: Example of a layout consisting of four shear wall groups.

Provided with the information above as well as the factored load for one specific shear wall group (discussed in Section 2.3.2), the nominal strength of each shear wall group in the layout shall satisfy the following equations [13]:

$$M_u \leq \phi M_n \tag{2.1}$$

$$P_u \leq \phi P_n \tag{2.2}$$

$$V_u \leq \phi V_n \tag{2.3}$$

where the subscripts u denote the factored load, M_u, P_u, and V_u are bending moment, axial load, and shear, respectively; and the subscripts n denote the nominal strength for bending moment, axial load and shear; ϕ is the reduction factor depending on the type of strength calculated.

Since the shape of each group is usually irregular, the standard formulas for reinforced concrete strength checks are no longer applicable. Reinforced concrete strength analysis must be developed starting with these assumptions, also illustrated in Figure 2-4:

- The strain in an embedded rebar is the same as that of the concrete around it. There should be no slip between these two materials.
- Plane cross sections will remain plane after loadings applied.
- The analysis is based on the stress-strain relationship and is subject to strength properties of these two materials whenever applicable.
- For computational simplicity, rebar set is deemed as one single uniformed plate with total area equals to that of the actual rebar set.
- Reinforced concrete members should be conservatively deemed as cracked members and an equivalent rectangular stress distribution is applied in the flexural strength calculation.

- The criterion for concrete compression failure is reaching maximum strain of 0.003.

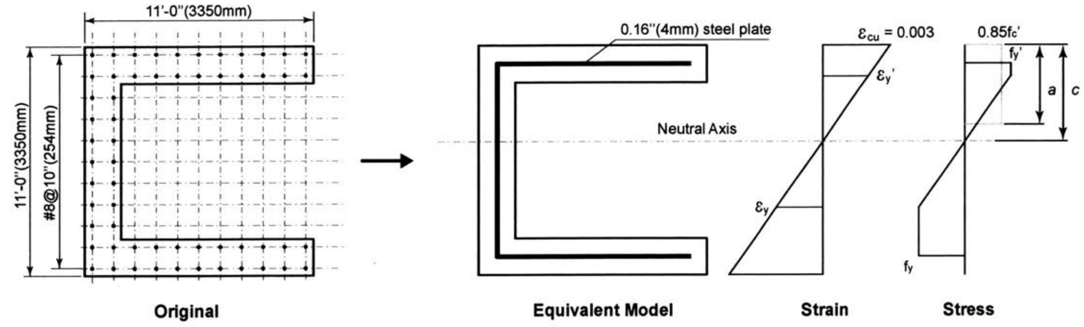

Figure 2-4: Equivalent model for reinforcement, and the stain and stress diagrams of the reinforced concrete under axial force and bending moment about the horizontal axis.

In the calculation of the strength of geometrically irregular concrete wall designs, failure shall fall into one of the following three situations, depending on the location of its neutral axis c under cracking:

- When c is close to the edge of compression side, the most exterior compression steel is below yield stress while the most exterior tension steel is at yield stress.
- When c is moderate, both the exterior compression steels are at yield stress.
- When c is close to the edge of tension side, the most exterior compression steel is at yield stress while the most exterior tension steel is below yield stress.

Based on the assumptions, the area of yielded sub-section either in tension or in compression can be calculated depending on c. Thus if c is given, the equations for force and moment equilibrium can be derived based on its yield situation:

$$\sum A_s' f_s'(d_s' - d_{cen}) + \sum A_c' f_c'(d_c' - d_{cen}) + \sum A_s f_s(d_s - d_{cen}) + M = 0 \qquad (2.4)$$

$$\sum A_s' f_s' + \sum A_c' f_c' + \sum A_s f_s + P = 0 \qquad (2.5)$$

where A_s', A_c', A_s are the section areas of steel in compression, concrete in compression and steel in tension, respectively; f_s', f_c', f_s are the stresses of steel in compression, concrete in compression and steel in tension, respectively. When rebar is in the yielding section, though its strain may vary, the stress remains at the yield stress $f_s' = f_y'$, $f_s = f_y$. And $(d - d_{cen})$ is the distance between the centroid of this sub-section and the cross sections centroid.

Whether a sub-section is in tension or compression depends on c. But since c is not knowable a priori, structural engineers, given P, usually obtain c from solving Eq. 2.5. For this case, however, shear wall designs with millions or billions of possible configurations make it difficult to derive a fixed form of these equations without knowing c.

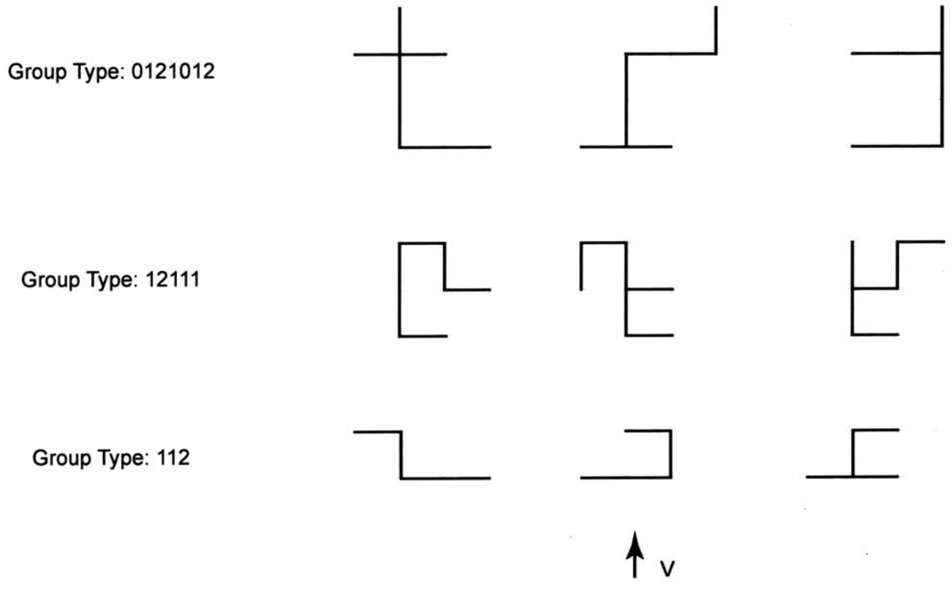

Figure 2-5: Example of three different shear wall configurations for a certain *GroupType*. Under the lateral load in the vertical direction, layouts in each row can be classified with a same string of numbers, which is referred to as *GroupType*.

Fortunately, this dilemma can be handled by simplifying the topological diversity of potential layouts and predefining and precomputing possible scenarios for c. Because of the plane-section assumption and strain-stress relationship (as shown in

Figure 2-4), the sub-sections at the same distance from the neutral axis have the same stress, regardless of their positions on the other direction.

For illustration, when considering lateral load in the vertical direction about a horizontal neutral axis, layouts in each row in Figure 2-5 can be classified with a same string of numbers, which is referred to as *GroupType*. Each digit represents the location of sub-section (*s*) and the value at this digit represents the number of sub-section (*s*) at this location.

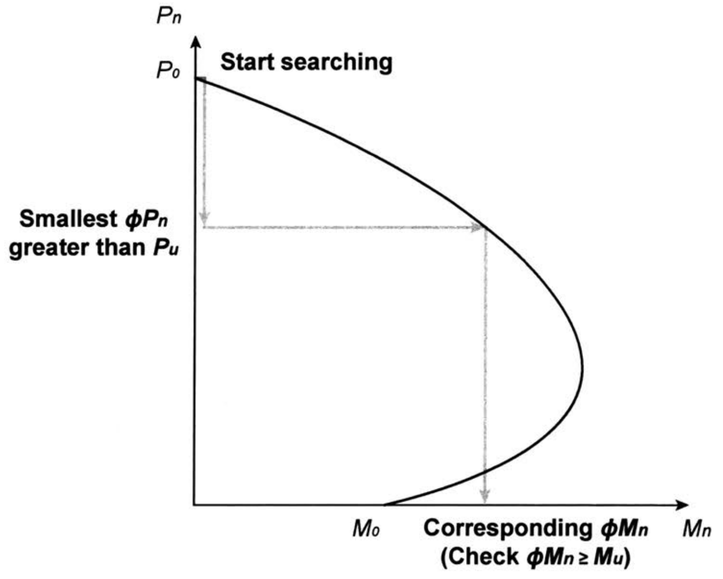
Pn
Po
Start searching
Smallest ϕPn greater than Pu
Mo
Corresponding ϕMn (Check ϕMn ≥ Mu)
Mn

Figure 2-6: Interaction diagram and the flexural strength checking process. The process starts with searching for the smallest ϕP_n greater than P_u (approximates $\phi P_n = P_u$); then the corresponding ϕM_n is found based on the diagram and then compared to M_u.

For each *GroupType*, every possible c falls into one of these subsection locations, and the M_n and P_n corresponding to this c can be calculated. Then P_n, with its corresponding M_n, constitutes an interaction diagram for this *GroupType* (Figure 2-6). Any data point on the left of this curve is considered a flexural failure. Computationally, while assuming $\phi P_n = P_u$, the flexural strength check can be achieved by comparing the ϕM_n corresponding to ϕP_n on this diagram to M_u. Either failing to find $\phi P_n = P_u$ or failing to satisfy $\phi M_n \geq M_u$ results in a failure in the flexural check. Implementation of this method is presented in Section 2.3.2.

2.3 Modified evolutionary algorithm

2.3.1 Description

In the twenty-story building example, a 6 by 8 grid has 110 edges. With each edge either activated or deactivated, the total number of all the possible shear wall layouts is 2^{110} (1.29×10^{33}). Optimizing through all the possibilities leads to a tremendous computational cost even with the help with the conventional evolutionary algorithm, which initiates with a random sample [12]. Due to the large number of possible layouts, a modified evolutionary algorithm is proposed in this thesis, which inherits the theory of natural selection propounded by Charles Darwin, but makes use of some biased and directional mutation and pairing methods to bias the evolution towards desired results.

Since the major goal is to minimize the structural weight, which, in this thesis, is evaluated by the number of shear wall members in the layout, optimized results will have fewer shear wall members. To further reduce the number of poorly scored layouts that are far from optimized and to increase the number of promising ones, this thesis forms a subset of all possible layouts in the 6 by 8 grid by constraining the maximum number of shear wall members (n_{max}) for the first generation. With algorithmic parameters customized by users, the system starts with a first generation randomly selected from the constrained subset and evaluates their performance both structurally and architecturally with a fitness or objective score (Fit). Then it ranks the layouts in the subset by their fitness scores, selects the individuals with highest fitness scores as parents and pairs these parents based on their properties. A group of offspring designs with the same size of the next generation (N_{gen}) shall be produced as a result of the breeding process (including crossover and mutation, described in

Section 2.3.4). Then those with the highest scores will be selected and become parents of the next generation. These steps will be repeated until it reaches either the last generation or the satisfying results, depends on the user. Figure 2-7 shows the flow chart of this process.

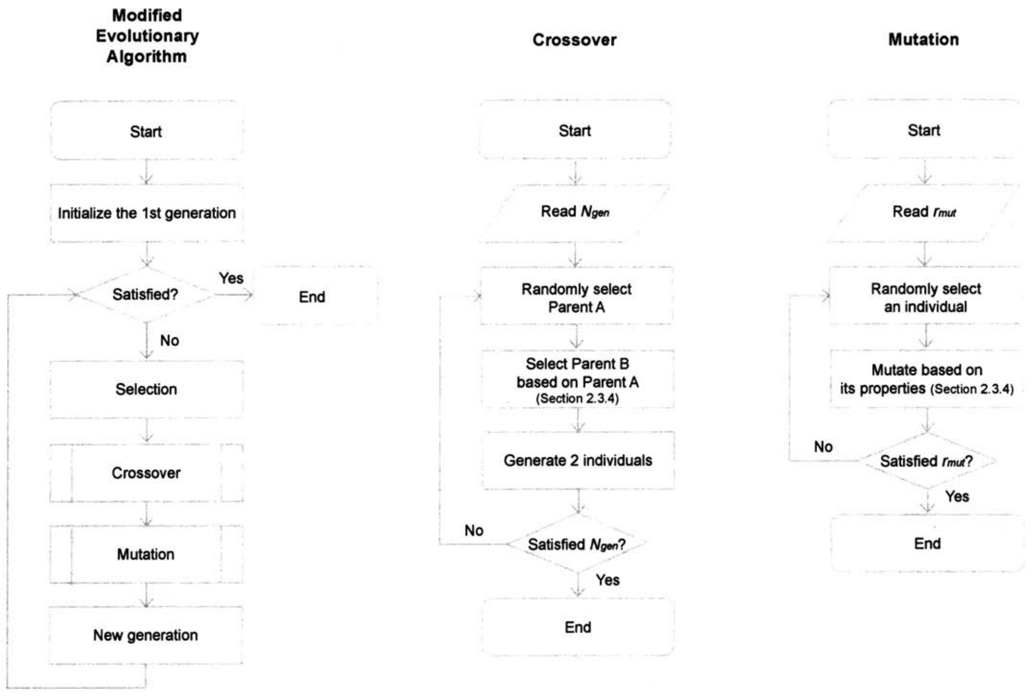

Figure 2-7: Flow chart of the modified evolutionary algorithm. The details of crossover and mutation process are presented in Section 2.3.4.

2.3.2 Objective function

The goal of this system is to minimize structural weight $W(\boldsymbol{v})$, which is a function of the shear wall layout ($\boldsymbol{v}$), subject to constraints on structural requirement (torsion, flexural, shear, and drift) and on basic architectural requirement (accessibility and necessary openings).

$$
\begin{aligned}
Minimize: \quad & W(\boldsymbol{v}) \\
s.t.: \quad & D_{cm\_cs} \leq D_{prefer} \\
& \phi V_n \geq V_u \\
& \phi M_n \geq M_u \\
& \phi P_n \geq P_u \\
& u \leq H/k \\
& N_{closed\_off} = 0
\end{aligned}
$$

D_{cm_cs} is the distance between center of mass and center of stiffness, D_{prefer} is the maximally permitted distance. M_u, P_u, V_u denote factored load moment, axial load and shear, while M_n, P_n, V_n denote nominal strength in flexural, axial load and shear, and ϕ is the reduction factor depending on the type of strength calculated. u is the horizontal drift, k is a factor for the drift limit and is usually taken as 500. H is the height of the building considered. N_{closed_off} is the number of closed-off shear wall groups (i.e. that contain architectural spaces that cannot be accessed) in this layout.

For implementation in an evolutionary algorithm, which cannot easily incorporate constraints, an unconstrained, penalizing objective function *f(v)* is introduced to evaluate the layouts performance:

$$f(\boldsymbol{v}) = (W(\boldsymbol{v}) + C_{total})(S_{weight} + C_{torsion} + C_{flexure} + C_{shear} + C_{drift} + C_{opening})^{\epsilon} \quad (2.6)$$

where $C_{torsion}$, $C_{flexure}$, C_{shear}, C_{drfit}, $C_{opening}$ are the constraint functions for torsional effect, flexural strength, shear strength, drift requirement, and openings, respectively (these are described in detail in the following paragraphs). S_{weight} is a special constraint function for structural weight. ϵ is the empirical factor and is usually taken as 2. C_{total} is a threshold constraint given by the following equation:

$$C_{total} = \begin{cases} 10000 & if \quad C_{torsion} + C_{flexure} + C_{shear} + C_{drift} + C_{opening} \geq 1 \\ 0 & if \quad C_{torsion} + C_{flexure} + C_{shear} + C_{drift} + C_{opening} < 1 \end{cases} \tag{2.7}$$

The value of *f(**v**)* with respect to a certain layout v is the fitness score Fit of this layout. A smaller fitness score indicates a better performance.

Based on the performance of the aspect examined, one of these three cases are applied to the unconstrained objective function *f(**v**)*:

- When the performance of layout v falls into the most preferable range (in terms of each individual constraint function), a zero value is assigned as the value of all penalty functions, and *f(**v**)* reduces to being the weight of the structural layout.
- When the performance falls out of the most preferable range, but within the acceptable range (in terms of each individual constraint function), the penalty function value usually follows a linear function, allowing its corresponding layout v to survive the selection but negatively affecting its performance evaluation results.
- When it violates a constraint and falls out of the acceptable range(in terms of each individual constraint function), the value of *f(**v**)* is considerably large, making its corresponding layout v impossible to survive the following selection process. For threshold constraints, theres no linear penalty function in the middle and the value of the penalty function is either 0 or very large.

The individual components of the objective function *f(**v**)* are defined as follows:

- S_{weight}: Special constraint function for structural weight
  Since this thesis focuses on the optimization of shear wall layout and there has to be a minimum amount of shear walls for lateral resistance, *f(**v**)* can never be

0. Although $W(\boldsymbol{v})$ already serves as a measurement of structural weight, there is no hierarchy between different numbers of shear wall members. Thus to further discourage exceeding a certain structural weight, this thesis introduces a special constraint that serves like a penalty factor, S_{weight}, which is benevolent to a certain range of structural weight (usually smaller than the preferred weight W_{prefer}) but aggressive to larger structural weight. S_{weight} can be calculated by the following equation:

$$S_{weight} = \begin{cases} 10 \times (W(\boldsymbol{v}) - W_{prefer}) & if \quad W(\boldsymbol{v}) \geq W_{prefer} \\ 0 & if \quad W(\boldsymbol{v}) < W_{prefer} \end{cases} \tag{2.8}$$

- $C_{torsion}$: Constraint function for torsional effect

Under lateral loadings such as wind loading or seismic loading, the torsional effect can be substantial, especially for tall buildings, and can cause severe failure in the structure. When a layout is no longer symmetric, the torsional effect can be mitigated by designing its center of stiffness to coincide with its center of mass [9]. Center of stiffness (x_{cs}, y_{cs}) and center of mass (x_{cm}, y_{cm}) are calculated by the following equations:

$$(x_{cs}, y_{cs}) = (\frac{\sum k_y x}{\sum k_y}, \frac{\sum k_x y}{\sum k_x}) \tag{2.9}$$

$$(x_{cm}, y_{cm}) = (\frac{\sum Ax}{\sum A}, \frac{\sum Ay}{\sum A}) \tag{2.10}$$

where k_x and k_y are the lateral stiffness of each shear wall in the x and y direction, respectively; A is the area of each shear wall member; (x, y) are the coordinates of the center of each shear wall member measured from the origin. With the distance between center of stiffness and center of mass D_{cs_cm} calculated by the following equation:

$$D_{cs\_cm} = \sqrt{(x_{cs} - x_{cm})^2 + (y_{cs} - y_{cm})^2} \tag{2.11}$$

the constraint function for torsional effect $C_{torsion}$ is given by:

$$C_{torsion} = \begin{cases} S_t \times D_{cs\_cm} & if \quad D_{cs\_cm} \le D_{prefer} \\ 1 & if \quad D_{cs\_cm} > D_{prefer} \end{cases} \tag{2.12}$$

where s_t is the objective function parameter indicating gradient for torsional effect, and is chosen by users; usually, $s_t \times D_{cs\_cm}$ is set to be in the range of $[0, 1]$; D_{prefer} is the preferred distance which is adjustable but is set as $1ft(30.5cm)$ by default. $D_{cs\_cm} > D_{prefer}$ will result in a large value of C_{total} (shown in Eq. 2.7), making Fit extremely large.

- $C_{flexure}$: Constraint function for flexural strength Since the computational method discussed in this thesis is only for the conceptual design phase, the load path for vertical loads and lateral loads is set as a parameter and is modifiable to users. By default, shear walls are assumed to resist the total lateral loads and half of the vertical loads. The bending moment and axial load resisted by a shear wall group is assumed to be proportional to its relative stiffness and relative area, respectively. Therefore, in one specific layout, the bending moment and axial load distributed to every shear wall group can be obtained with the following equations:

$$M_{uj} = \frac{M_{total} I_j}{\sum I_j} \tag{2.13}$$

$$P_{uj} = \frac{P_{total} A_j}{\sum A_j} \tag{2.14}$$

where M_{uj} is the overturning moment distributed to a shear wall group, M_{total} is the factored overturning moment on a layout, I_j is the moment of inertia of this shear wall group; P_{uj} is the axial force distributed to a shear wall group, P_{total} is the factored axial force on a layout, A_j is the area of this shear wall group; and the subscripts j denotes a shear wall group.

Generally, shear walls subjected to combination of axial load and flexure should be designed as compression members [9]. The flexural strength check in shear walls of tall buildings is critical and its calculation is very complex, especially for the irregularly shaped shear wall groups in this thesis. For one specific irregularly shaped shear wall group, its neutral axis varies with the load applied on the shear wall group. And the load is a variable depending on the relevant stiffness. The steps for flexural check, given P_{uj} and M_{uj}, have been presented in Section 2.2.3 so the following content is mainly focused on the implementation of the flexural check. Considering the efficiency of the system, this thesis proposes to pre-calculate every possible neutral axis c and its corresponding M_n, P_n, and I_{cr} for every possible *GroupType* smaller than a 3 by 3 grid, and store these data in a spreadsheet. The 3 by 3 grid is chosen because it already has more than 10,000 possibilities, close to but under the maximum storage capacity of the spreadsheet. With these data calculated in advance, the optimization system can search for the results and consequently it is less time-consuming. Those *GroupTypes* not included in the spreadsheet (larger than 3 by 3 grids) are calculated in real time along the optimization process. This thesis uses I_{cr}, the moment of inertia under cracking, in the calculation of drift. For each shear wall group, according to Section 2.2.3, if the position of neutral axis (c) is determined, the corresponding I_{cr} can be calculated and then stored as one of the properties under this c.

More specifically, in the work presented in this thesis, the spreadsheet is transformed to a hash map (dictionary) in Python with *GroupType* being the key. During optimization, every newly-generated shear wall group is assigned four *GroupTypes* in four different directions respectively. Thus, from the hash map, a list of every possible neutral axis location c under cracking and its corresponding moment and axial load is located based on the *GroupType* in each direction. According to Section 2.2.3, the process starts from picking one direction, locating the *GroupType* corresponding to this direction in the hash map, and searching for the smallest ϕP_n greater than P_u (approximates $\phi P_n = P_u$). This P_n, if successfully found, would serve as a key for retrieving corresponding c, M_n, and I_{cr}. Flexural strength check for this shear wall group in this direction passes if $\phi M_n \geq M_u$. Otherwise, either failing to find $\phi P_n = P_u$ (axial failure) or $\phi M_n < M_u$ (bending failure) results in a failure in flexural check in this direction.

Since the geometry of each shear wall group is not necessarily symmetric, flexural strength check should be conducted for every possible direction (for a rectangular building, there are usually four directions). If and only if all the shear wall groups in one layout pass the flexural check in every direction, shall this layout be deemed as satisfactory in terms of flexural strength. Thus the constraint function for flexural strength $C_{flexure}$ can be obtained by:

$$C_{flexure} = \begin{cases} 0 & if \quad pass \\ 1 & if \quad fail \end{cases} \tag{2.15}$$

A flow chart for flexural check is presented in Figure 2-8.

- C_{shear}: Constraint function for shear strength

  Structural failure due to shear may be less dominant for shear walls in this example building for its large height-to-length ratio. Out of general consideration on other possible dimensions and height that could be designed by users in future, this thesis still conducts a simplified method for a basic shear strength check. Shear walls are assumed to resist the total lateral loads, and shear forces in one direction are only resisted by shear wall members in this direction. Since shear taken by a shear wall member is based on its relevant area in its direction, it would be sufficient to check shear strength of one shear wall member with a rectangular geometry. Shear distributed to a shear wall member can be obtained by:

$$V_{uj} = \frac{V_{total} A_j}{\sum A_j} \tag{2.16}$$

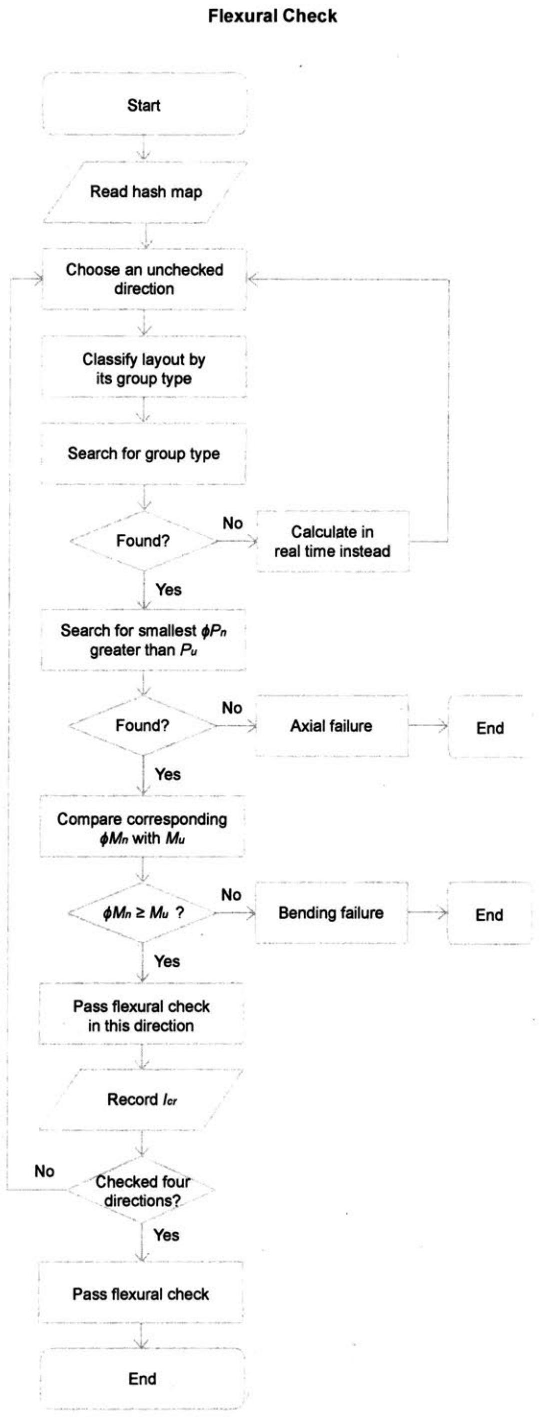
Flexural Check
Start
Read hash map
Choose an unchecked direction
Classify layout by its group type
Search for group type
Found?
No
Calculate in real time instead
Yes
Search for smallest ϕP_n greater than P_u
Found?
No
Axial failure
End
Yes
Compare corresponding ϕM_n with M_u
$\phi M_n \geq M_u$?
No
Bending failure
End
Yes
Pass flexural check in this direction
Record I_{cr}
No
Checked four directions?
Yes
Pass flexural check
End

Figure 2-8: Flow chart for flexural check.

where V_{total} is the factored total shear force in one direction (the one at the base building is critical, also called base shear), V_{uj} is the factored shear force in this direction and A_j is the area in this direction of one specific shear wall group.

According to ACI [1], given V_{uj} as well as the dimension of concrete and the reinforcement, shear strength provided by concrete is calculated by:

$$\phi V_c = \phi 2\sqrt{f_c'}h0.8l_w \tag{2.17}$$

The shear strength provided by horizontal reinforcement is calculated by:

$$\phi V_s = \phi \frac{A_v f_y d}{s_2} \tag{2.18}$$

And their factored sum should be no larger than to Vuj [9]:

$$\phi V_n = \phi V_c + \phi V_s \geq V_{uj} \tag{2.19}$$

where A_v is the total area of the horizontal shear reinforcement within a distance s_2, $\phi = 0.75$, $f_y = 60ksi(414MPa)$ and d is the effective horizontal length of shear wall.

Since the geometry of each shear wall group is not always symmetric, a shear strength check should be conducted for every possible direction (for a rectangular building, there are usually two directions). If and only if all the shear wall groups in one layout pass the shear check in each direction, shall this layout be deemed as satisfactory in terms of shear strength. This constraint is given by the following equation:

$$C_{shear} = \begin{cases} 0 & if \quad pass \\ 1 & if \quad fail \end{cases} \tag{2.20}$$

- C_{drift}: Constraint function for wind drift

  For tall buildings, inter-story and overall lateral deflections are expected to remain within acceptable range. Thus in serviceability design, wind drift limit is adopted to relieve motion perceptibility, to reduce damage to the structural and non-structural members, and to alleviate P-Delta effects caused by the displaced gravity load.

  There are two major contributions to the drift of a building: bending and shear. The total drift is the sum of drift due to bending ($u_{bending}$) and drift due to shear (u_{shear}):

$$u = u_{bending} + u_{shear} \tag{2.21}$$

  And since a tall building is modeled as a cantilever, $u_{bending}$ and u_{shear} in each direction shall be calculated as follows:

$$u_{bending} = \frac{Q_u H^3}{3EI_{cr}} \tag{2.22}$$

$$u_{bending} = \frac{Q_u H}{\frac{5}{6} G A_{total}} \tag{2.23}$$

  where E is the elastic modulus of concrete and H is the total height of this building, I_{cr} is considered for bending rigidity in the examined direction under cracking, G is the shear modulus of concrete, A_{total} is the total shear wall area of this building, 5/6 is the shear factor [9].

  Since there is no fixed criterion for wind drift limit, users are expected to define their criterion in advance (e.g. $u \leq H/500$). However, only if the flexural check is satisfied by every shear wall group in the layout shall this layout proceed to check for serviceability. Like the shear strength check and the flexural strength check, the wind drift check should also be conducted for every possible direction. And if the drift check fails for a layout in any direction, this layout is assumed failing the drift check. Thus the constraint function for wind drift is defined as follows:

$$C_{drift} = \begin{cases} 0 & if \quad u \leq H/500 \\ 1 & if \quad u > H/500 \end{cases} \tag{2.24}$$

- $C_{opening}$: Constraint function for accessibility and openings

  A practical shear wall group should provide proper accessibility and necessary openings. Thus spaces closed off on all sides should be avoided for the shear wall groups. This thesis introduces a constraint function $C_{opening}$ which provides this limit by controlling the ratio of number of different coordinates (n_{dif_coord}) and number of members (n_{mem}) in a shear wall group, r_{coord}:

$$r_{coord} = \frac{n_{dif\_coord}}{n_{mem}} \tag{2.25}$$

  Thus the constraint function for accessibility and openings is defined by the following equation. There would be no penalty added to the objective function if there is no enclosed shear wall group in the layout.

$$C_{opening} = \begin{cases} 0 & if \quad r_{coord} > 1 \\ 1 & if \quad r_{coord} \leq 1 \end{cases} \tag{2.26}$$

2.3.3 Selection and diversity filter

Once all the individuals in one generation are evaluated in terms of performance, they are given a fitness score property Fit. The selection stage imitates the natural selection by allowing only the best performing individuals to survive and thus become the "parents"of the next generation. The number of parents (N_p) in each generation (N_{gen}) can be calculated as follows:

$$N_p = N_{gen} \times r_p \tag{2.27}$$

where parent ratio (r_p) is defined by users and a smaller ratio gives a harsher survival challenge to the current generation.

These parents, which are relatively a small portion of the prior generation, are responsible for generating all the individuals as many as one generation. This breeding procedure, including crossover and mutation, aims to help inherit the advantages of parents while increasing the diversity of the offspring (see Section 2.3.5). Thus it is imperative that each pair of mating parents is very likely to produce some individuals that have similar topologies, and thus similar scores. Some of these similar individuals might have low scores and thus cannot be selected as parents for the next generation, but some of them can rank among the best and thus reduce the diversity of the next generation. In nature, best performing individuals, despite their similarity, should survive without bias; however, since the size of domain (all the possible layouts) in this system far exceeds the size of generations, the proposed method encourages diversity so that a wider range of good solutions are more likely to be found. Therefore, a technique called a diversity filter is introduced, which filters out the similar layouts during selection based on the threshold value of diversity ratio ($r_{div\_threshold}$) set by users. Two layouts are deemed as similar when their actual diversity ratio (r_{div}) is larger than $r_{div\_threshold}$. Diversity ratio (r_{div}) is defined by the following equation:

$$r_{div} = \frac{n_{overlap}}{n_a ver} \tag{2.28}$$

where $n_{overlap}$ is the number of members existing in both layouts; n_{aver} is the average number of members in two layouts.

During computational implementation of this method, after ranking all the individuals in the current generation by their performance and saving them in a "candidate list", a "survivor list" initiated with the best individuals is created. For every individual in the candidate list, compare it with the individual(s) in the survivor list until one candidate, different enough (based on r_{div}) from every individual in the survivor list, appears. This candidate is added to the survivor list and thus deleted from the candidate list.

2.3.4 Crossover and mutation

Conventional evolutionary algorithms, which use randomness during the crossover and mutation procedure, introduce both beneficial and detrimental changes to the offspring. Thus these algorithms usually move around the domain in every direction. Consequently, optimized results cannot be guaranteed without considerably large number of generations. Unlike conventional evolutionary algorithms, the proposed method in this thesis moves towards the optimum by allowing the algorithm to pair parents and guide the crossover and mutation process based on *Locationtype* and $Modificationtype$.

Table 2.3: Definitions, properties and corresponding spouse of each $Modificationtype$.

$Modif.type$	Structural Performance S	Structural Weight $W(v)$	Properties	Spouse's $Modif.type$
Major	Fail	$> W_{prefer}$	Has too many walls, still fails structural performance; Need: major modification in both aspects	Minor
Major+	Fail	$\leq W_{prefer}$	Although layout has few walls, Fails structural performance; Need: more walls to pass the strength check.	Major-
Major-	Pass	$> W_{prefer}$	Although layout passes structural performance, Has too many walls; Need: reduce walls to achieve the goal	Major+
Minor	Pass	$\leq W_{prefer}$	Passes structural performance; Has few walls; Need: preserve its good features	Minor

Every layout, based on the number of shear walls and structural performance, can be classified as one of the following $Modificationtypes$ defined in Table 3. The structural performance (S) of this layout can be either "Pass" if all the strength checks and drift check are passed, or "Fail" if there exists failure in drift check or at least one strength check failing in at least one direction; W_{prefer} is the preferred structural weight, which is a prediction of the optimal structural weight and can be selected with the method described in Section 2.4.

And according to the relevant location of center of stiffness and center of mass, it can be classified as one of the following *Locationtypes*:

$$Location\ type = \begin{cases} 1st\ quadrant & if \quad x_{cs} > x_{cm}\ and\ y_{cs} \geq y_{cm} \\ 2nd\ quadrant & if \quad x_{cs} \leq x_{cm}\ and\ y_{cs} > y_{cm} \\ 3rd\ quadrant & if \quad x_{cs} < x_{cm}\ and\ y_{cs} \leq y_{cm} \\ 4th\ quadrant & if \quad x_{cs} \geq x_{cm}\ and\ y_{cs} < y_{cm} \\ Centered & if \quad x_{cs} = x_{cm}\ and\ y_{cs} = y_{cm} \end{cases} \tag{2.29}$$

Once selection based on fitness score is conducted, the "survivors" after selection will become parents for the next generation. The general process starts with randomly picking one individual from these "survivors" and pairing it with another individual layout with diagonal location (diagonal quadrant) and corresponding spouses $Modification type$. Four examples are shown in Figure 2-9.

Next, the binary string chromosome of each parent is split at the same random point and recombined. The reassembled chromosome becomes the chromosome of a newly-generated individual. It is worth noting that the $Modification type$ and *Locationtype* are calculated whenever a new individual is generated. Then another pair of parents is picked and goes through the same procedure until the number of individuals generated reaches the size of generation.

Mutation is encouraged to introduce potentially beneficial changes to the upcoming generation. Mutation rate (r_{mut}) is set by users and is the ratio between number of mutated individuals (N_{mut}) and number of all the individuals (N_{gen}) in the generation. Individuals are randomly selected from a new generation as mutation candidate. Since each newly-generated individual has $Modification type$ and *Locationtype*, each

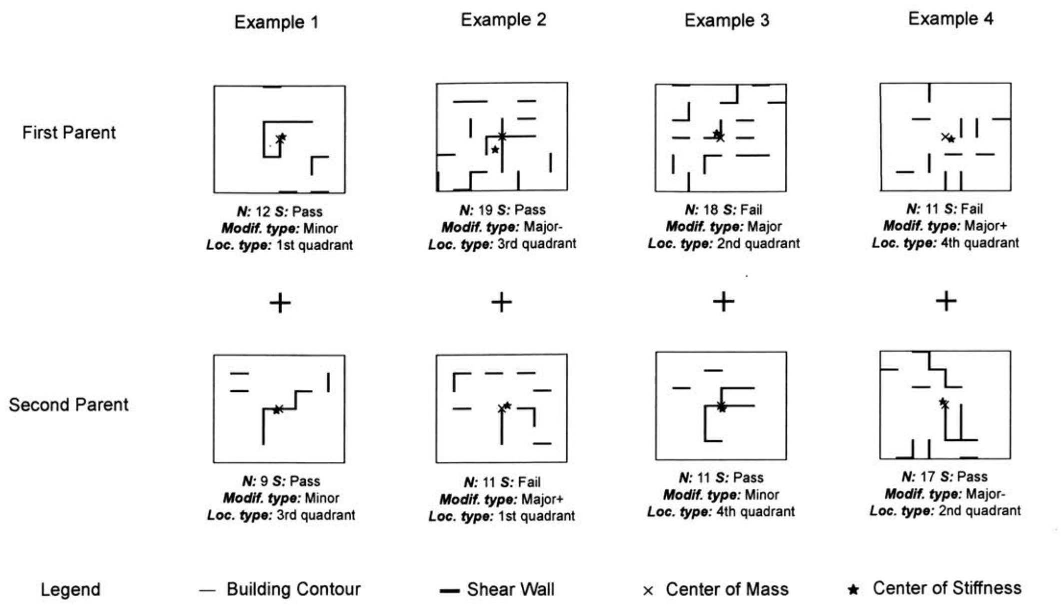

Figure 2-9: Four examples of the pairing process during crossover.

mutation candidate is mutated by either adding shear wall members or reducing shear wall members based on the its $Modificationtype$. To better reduce the torsional effect, the system selects the shear wall members with the same quadrant to be turned off, and those with the diagonal quadrant to be turned on.

2.4 Parameter selection

Apart from the evolutionary parameters (such as mutation rate, generation size, and number of evolutions), the quality of the results also varies significantly with algorithmic parameters related to objective function, especially when there are multiple variables in this function. Since the significances of these variables are considerably correlated with each other, the result of one specific parameter set is less predictable without tedious experiments and complex analyses. To improve the performance and efficiency of this optimization system, a simplified method for parameter selection is proposed and can be implemented either before or during the optimization process.

Since the objective is to minimize the structural weight, the most significant parameters are the preferred structural weight W_{prefer} and the maximum number of shear wall members for the first generation n_{max}. Thus adjusting these two parameters is sufficient for the optimization in this research. The general process of selection is: after setting up the case, run the system several times and each time slightly reduce W_{prefer} and n_{max} ($W_{prefer} < n_{max}$) until a failure in the strength check occurs in the top five individuals of the first generation. For example, initialize W_{prefer} to be $0.5n_{mem}$ and n_{max} to be $0.3n_{mem}$. And each time deduct $0.05n_{mem}$ from both parameters until the failure occurs. If reduction continues, an increasing number of failures in the strength check will appear in the top five individuals of every generation. Thus the occurrence of first failure is the critical point. This indicates that the system is trying excessively to optimize the structural weight by placing too much significance on this factor that it fails to filter out a lighter structure with unacceptable structural strength. The values of W_{prefer} and n_{max} at this critical point shall be deemed as the optimal values, which are most likely to yield the optimum structural weight.

Chapter 3

Evaluation of the methodology

The 6 by 8 grid example is used in this chapter to evaluate the effectiveness of the methodology presented in this thesis. Although parameter values of the algorithm should be set in advanced by users based on the objectives and requirement, in this example, the parameter setup is shown in Table 3.1.

Table 3.1: Parameters for the general case study.

Size of generation N	Num. of generations N_{gen}	Mutation rate r_{mut}	Parent ratio r_p	Max num. of walls n_{max}	Preferred weight W_{prefer}	Preferred distance D_{prefer}
2000	6	0.3	0.1	$0.3n_{mem}$	$0.1n_{mem}$	1ft (30.5cm)

For a problem with the parameters mentioned above, one complete optimization process takes around two minutes on a standard laptop. Figure 3-1 illustrates the top five individual layouts in the first generation, the third generation and the last (sixth) generation. Obviously, the algorithm does improve the performance along each generation. To prove this conclusion in a more objective and quantitative way, Figure 3-2 plots the mean, median, maximum, minimum of (a) the fitness scores; (b) number of shear walls (evaluating structural weight); (c) distance between center of stiffness and center of mass for the top ten layouts in each generation. According to the figure, across the generations that minimize the value of unconstrained objective function *f(**v**)*, structural weight gradually decreases and is about to converge after 4 generations. The distance between center of stiffness and center of mass D_{cs_cm}, after a drastic drop in the first evolution, converges quickly and remains within 1ft, which is the preferred distance D_{prefer}, indicating the high efficiency of this algorithm.

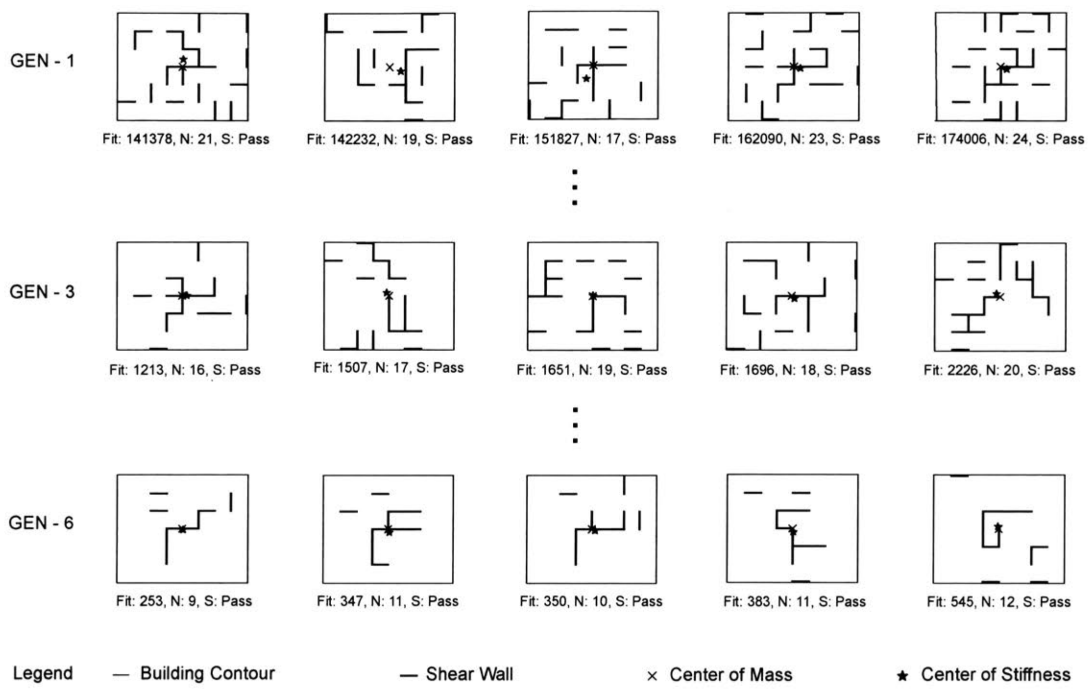
GEN - 1
Fit: 141378, N: 21, S: Pass
Fit: 142232, N: 19, S: Pass
Fit: 151827, N: 17, S: Pass
Fit: 162090, N: 23, S: Pass
Fit: 174006, N: 24, S: Pass
GEN - 3
Fit: 1213, N: 16, S: Pass
Fit: 1507, N: 17, S: Pass
Fit: 1651, N: 19, S: Pass
Fit: 1696, N: 18, S: Pass
Fit: 2226, N: 20, S: Pass
GEN - 6
Fit: 253, N: 9, S: Pass
Fit: 347, N: 11, S: Pass
Fit: 350, N: 10, S: Pass
Fit: 383, N: 11, S: Pass
Fit: 545, N: 12, S: Pass
Legend
— Building Contour
— Shear Wall
× Center of Mass
★ Center of Stiffness

Figure 3-1: The top five individual layouts in the first generation, third generation, and last (sixth) generation. Beneath each layout, the objective function score is given (Fit), along with the number of shear walls or equivalently structural weight (N) and whether the strength checks are satisfied (S).

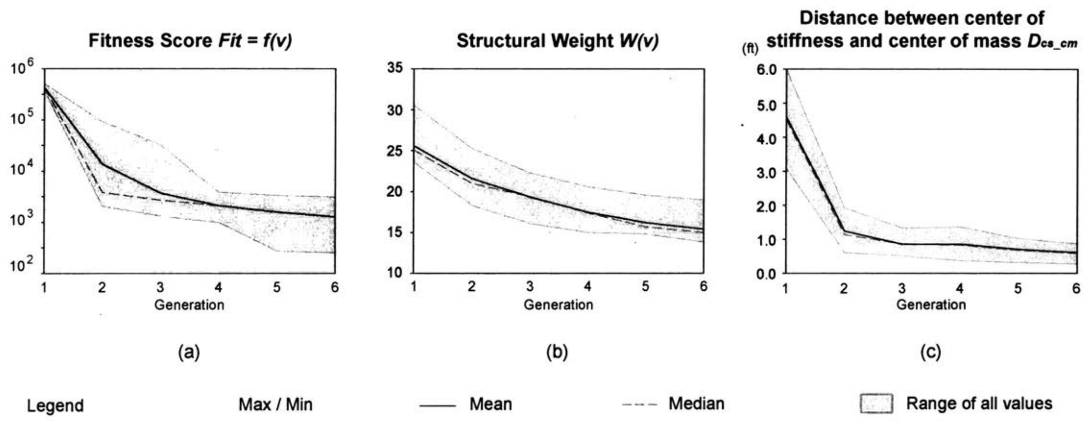

Figure 3-2: Plots of the various metrics of structural performance of the top ten designs across six generations of the evolutionary algorithm.

Chapter 4

Complex cases

4.1 Void space (Structures with plan irregularities)

Footprints of buildings considered in this system were by default rectangular, as studied above. However, if the building of interest has an irregular contour (such as L shape or Y shape) or atriums, or if the program requires some spacious area without any walls, an extending concept of void space can be applied.

In the proposed method, every layout is transformed to a binary number with each digit representing a line segment in the grid and each line segment has features of coordinates. Thus by introducing void space which contains the location information of these void areas (irregular contour, atriums, or wall-free areas), the system always deactivates those line segments that fall within these void spaces and thus eliminating shear walls from these areas. One thing worth-noting is that for irregular contour and atriums, the areas of void spaces should be considered non-existing and shall never be involved in loading calculations as well as penalty on torsional effect. For wall-free areas, nevertheless, the areas of void spaces are wall-free areas on the slab that have volume and are able to bear load.

This thesis illustrates the evolution of the structures with void spaces in Figure 4-1. Wall-free areas at three corners of this hshaped footprint encourage concentrating the shear walls in the center. The significant improvement in terms of structural weight is suggested by a variety of layouts in each generation.

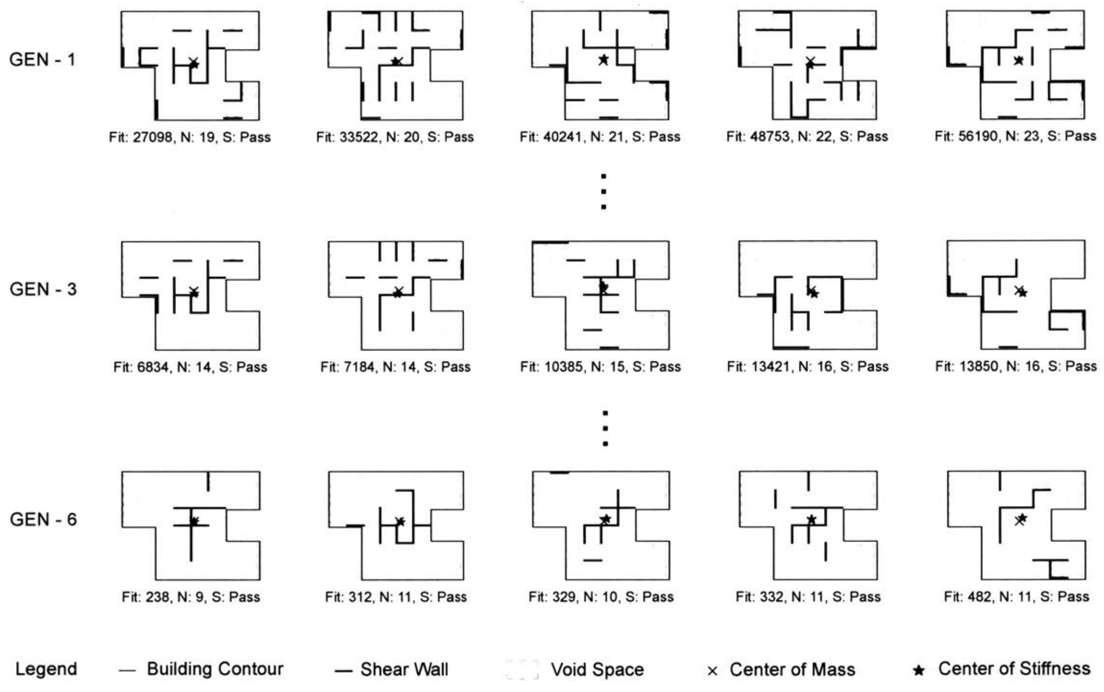
GEN - 1
Fit: 27098, N: 19, S: Pass
Fit: 33522, N: 20, S: Pass
Fit: 40241, N: 21, S: Pass
Fit: 48753, N: 22, S: Pass
Fit: 56190, N: 23, S: Pass
GEN - 3
Fit: 6834, N: 14, S: Pass
Fit: 7184, N: 14, S: Pass
Fit: 10385, N: 15, S: Pass
Fit: 13421, N: 16, S: Pass
Fit: 13850, N: 16, S: Pass
GEN - 6
Fit: 238, N: 9, S: Pass
Fit: 312, N: 11, S: Pass
Fit: 329, N: 10, S: Pass
Fit: 332, N: 11, S: Pass
Fit: 482, N: 11, S: Pass
Legend
Building Contour
Shear Wall
Void Space
Center of Mass
Center of Stiffness

Figure 4-1: The top five individual layouts in the first generation, third generation, and last (sixth) generation for an h-shaped footprint with corner wall-free areas. Both fitness score and structural weight drop sharply along the evolutions.

In Figure 4-2, different wall-free areas with the same irregular contour demonstrate the application, with or without a conventional structural core, under different programmatic requirement (such as the main usage of the building). For example, an exterior void encourages the formation of a structural core and thus is more preferable for an office building, while interior void, such as Hshaped void encourages spread-out shear wall placement and is thus more suitable for a residential building. As is suggested in Figure 4-2, minimum structural weight varies with the configuration of void spaces. Thus choosing unreasonable void spaces can result in a waste of material.

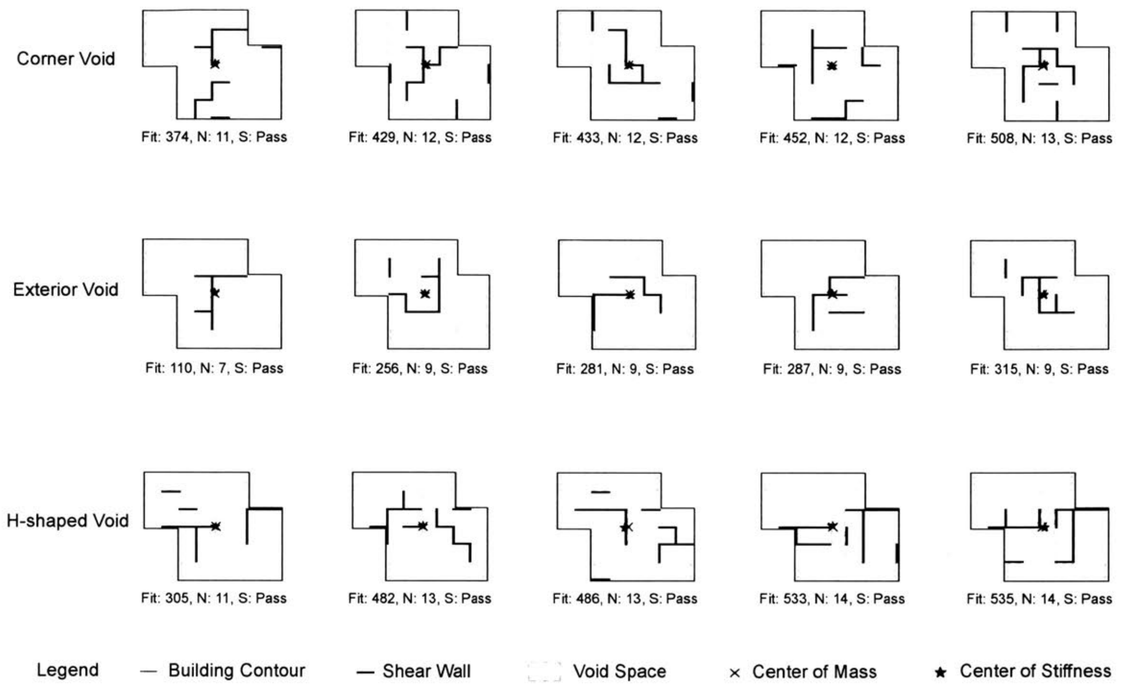

Figure 4-2: Result sample of layouts with different wall-free areas and the same irregular contour.

To show the diversity and potential of this system, Figure 4-3 introduces three types of layouts with distinct contours as well as void spaces. This figure also shows that different initial footprints (including contour and void spaces) are likely to end up with distinct optimal shear wall layouts.

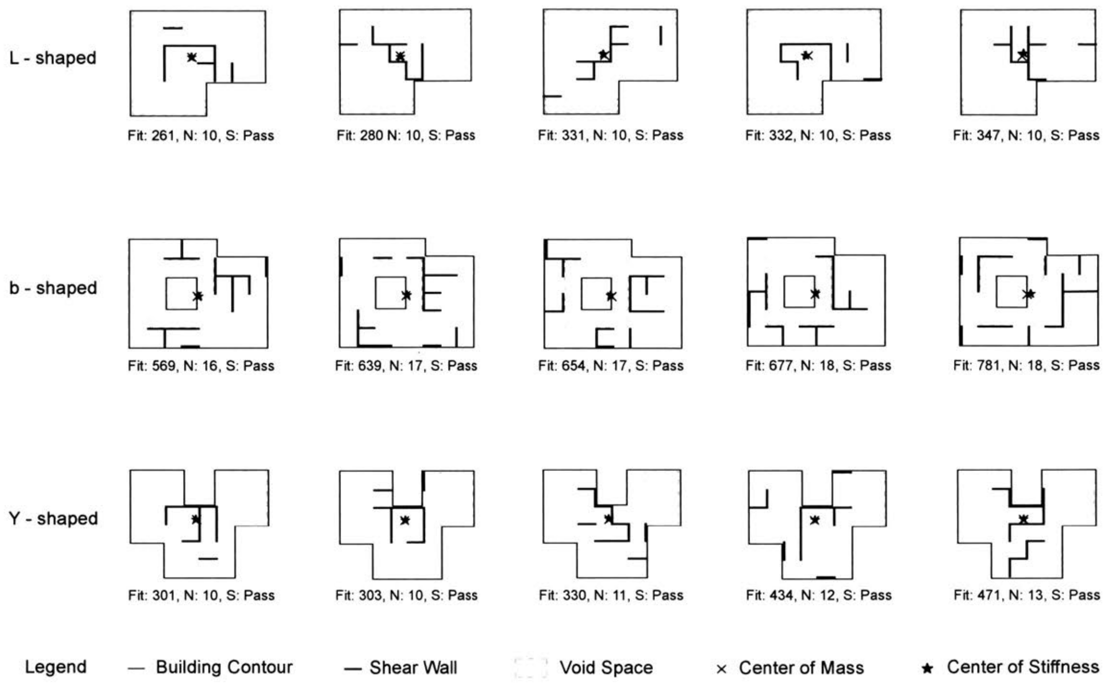

Figure 4-3: Result sample of layouts with different wall-free areas and the different irregular contours. The square in the center of the b-shaped footprint indicates an atrium.

4.2 Fixed floor plans

Although the basic method described in Section 2.3 can provide various types of layouts for a flexible conceptual design, an architect sometimes already has a floor plan in mind and thus only needs help arranging the placement of shear walls within it (Figure 4-4a). This section gives a brief introduction on how the basic method can be easily transformed to apply to this special case.

While the basic method models initiate the ground structure as rectangular grids, the transformed version can still use the ground structure concept but changes the original quad mesh containing all the orthogonal edges to a subset of these edges which lie along a wall (structural or non-structural) existing on the floor plan (Figure 4-4b). The binary number can still be used and each digit represents a line segment in the subset. The following evolutionary algorithm procedures remain the same as those described in Section 2.3.

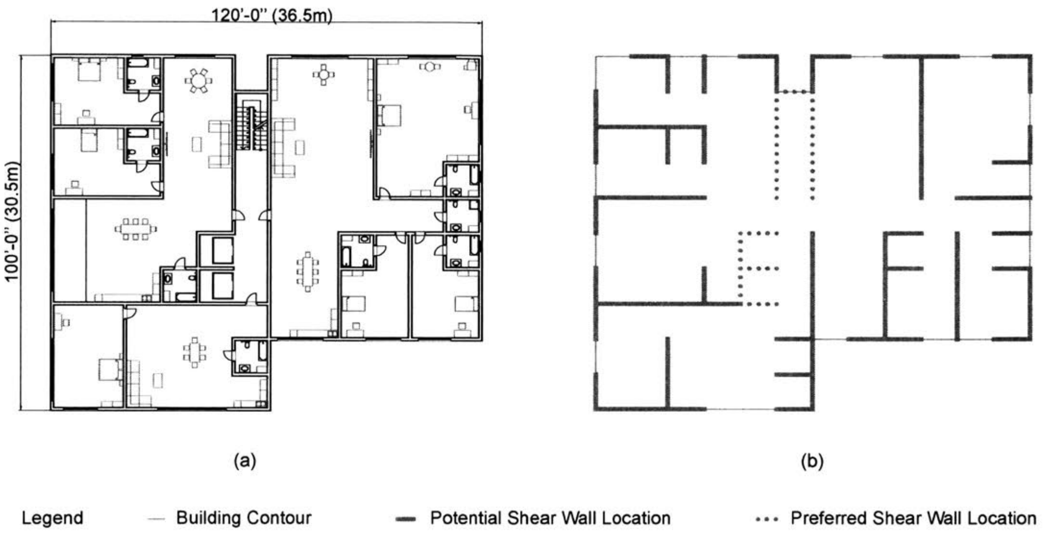
120'-0" (36.5m)
100'-0" (30.5m)
(a)
(b)
Legend
— Building Contour
▬ Potential Shear Wall Location
••• Preferred Shear Wall Location

Figure 4-4: Example of the fixed floor plan and its corresponding model in the system: (a) the floor plan which is designed by an architect beforehand and delivered to the structural engineer, requiring placement of shear walls; (b) the model interpreted by the system corresponds to the original floor plan.

In practice, elevator cores and walls around the staircases are often ideal places for placing shear walls not only because of their vertical continuity throughout the building but also for providing fireproof structure around emergency egress. Thus in the fixed floor plan extension, users can pick several walls as preferred shear wall locations. In the mutation process, if any of these selected walls are inactive, the system will show preference on activating it. A simple way to manage this is: first, set a threshold z0 within [0,1], where 0 corresponds to no preference and 1 corresponds to a highly desired shear wall location. Then before mutation, randomly pick a real number z from 0 to 1, if z is smaller than z0 , then activate an inactive shear wall member (if any) randomly selected from the preferred shear wall locations. Do this only one time for each mutation process.

Figure 4-5 illustrates the optimization process for the floor plan proposed in Figure 4-4. While reduction on the structural weight is less significant, it is effective in minimizing the torsional effect. However, the methodology has an impressive advantage in computational cost. On a standard laptop, it takes three to five minutes to go through the evolution and generate one set of diverse, highperforming results for layouts with the dimension 36m by 30m.

If building codes require egresses (e.g. elevators or stairs) to be at a certain places or within a certain distance from each other or from a center location, some shear wall members can be prelocated; in implementation, the edges of these shear walls members will always be activated in every layout.

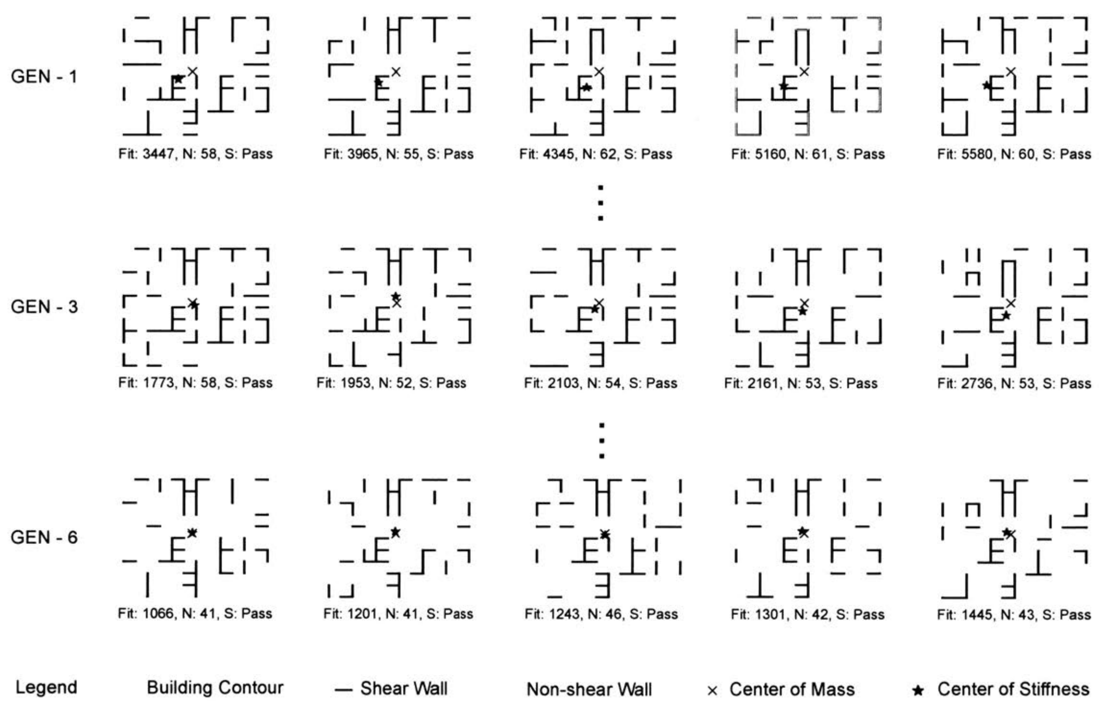
GEN - 1
Fit: 3447, N: 58, S: Pass
Fit: 3965, N: 55, S: Pass
Fit: 4345, N: 62, S: Pass
Fit: 5160, N: 61, S: Pass
Fit: 5580, N: 60, S: Pass
GEN - 3
Fit: 1773, N: 58, S: Pass
Fit: 1953, N: 52, S: Pass
Fit: 2103, N: 54, S: Pass
Fit: 2161, N: 53, S: Pass
Fit: 2736, N: 53, S: Pass
GEN - 6
Fit: 1066, N: 41, S: Pass
Fit: 1201, N: 41, S: Pass
Fit: 1243, N: 46, S: Pass
Fit: 1301, N: 42, S: Pass
Fit: 1445, N: 43, S: Pass
Legend
Building Contour
— Shear Wall
Non-shear Wall
× Center of Mass
★ Center of Stiffness

Figure 4-5: The top five individual layouts in the first generation, third generation, and last (sixth) generation for a fixed floor plan.

Chapter 5

Conclusion

5.1 Summary of contributions

This thesis has proposed a new methodology produces diverse, high-performing shear wall layout designs for tall buildings that can respond to both structural and architectural design goals. The method is compatible with a large variety of buildings, from low-rise to high-rise, from wide to tall (aspect ratio), from office to residential, and from box to irregularly-shaped. Furthermore, it can be incorporated flexibly either before or after the design of architectural floor plan, sparking new inspiration or conforming to an agreed upon system. Integrating structural performance and architectural design, the diverse optimized results not only provide designers with a wide range of distinct layouts to choose from, but also pre-calculated the structural performance, ensuring that any layouts selected from this subset are among the best-performing ones. Once a conceptual design for the shear wall layout is selected, it can be analyzed and detailed much more precisely by the structural engineer later in the design process.

This thesis also proposes general solutions for unaddressed problems. With respect to structural performance, a simplified auto-calculation system for reinforced concrete design has been established and applied in this research, which saves computational time and memory for the optimization process, and addresses the blank in the design codes regarding irregularly configured shear walls.

Moreover, in terms of the optimization algorithm, this thesis presents customizations of the conventional ground structure as well as the evolutionary algorithm. Since the population containing all possible layouts is extremely large, which requires huge computational cost under the conventional evolutionary algorithm, the modified version encourages inheriting advantageous features and triggers beneficial changes to each generation. It sharply reduces the computation time to be less than one-tenth of that for an average calculation without these modifications.

Lastly, this thesis aims to fill the gap between engineers and architects, and reduce the trial-and-error design process for buildings so that better and more integrated solutions can be found.

5.2 Future work

The work presented in this thesis focuses on optimization in terms of architectural design and structural performance. However, with the complexity of architectural considerations, the non-fixed versions may not yield plausible layout in terms of, for example, space arrangement. For instance, the optimized results may indicate an elevator well in an apartment or a living room right beside the door of the apartment. Further, this thesis only considers shear wall layout in plan-view, which ignores the variation in dimension for sections along the height of the building. Algorithm-wise, the objective function contains many variables which should be analyzed and generalized.

Thus, future improvements include the spatial arrangement and connectivity to consider human usability and convenience, exploration in three dimensions to widen the application of this research, and analysis and improvement on the evaluation criteria to make this algorithm more general, comprehensible, and compatible.

Moreover, since some engineers cannot easily access the Python-based computational system proposed in this thesis, further development of a computational tool or software with a graphical user interface is suggested as part of the future work.

Appendix A

Scripts

There are mainly two types of scripts involved in this thesis: one type is for generating the hashmap to store the interaction diagram for all the configuration within a 3 by 3 grid; and another type of script is the main loop for the modified evolutionary algorithm. This appendix will show 2 scripts: A.1 Script or generating concrete strength, and A.2 Script for the fixed floorplan extension case. If the user are considering the unfixed cases, just deactivate the pre-select function, activate the quardrant function, and input the altrim/wall-free area, if any. The scripts are writing for Python 2.7.

A.1 Script for generating concrete strength

```
###### This script is writen by Yu Zhang at MIT in July 2016
###### and is designed for generating the cracking depth and cracked inertia
###### for all the possible shear wall configuration in a 3 by 3 grid

import xlsxwriter
import math
import csv
import xlsxwriter
from pyDOE import *
import copy
import pickle

#####PARAMETERS

###algorithm parameters

dict_all = {}

n_grid_x = 3
n_grid_y = 3 #grid size: n_grid_x by n_grid_y
num_zone = n_grid_y*2+1

concrete_strength = 5 #ksi
```

```
steel_strength = 60#ksi

thick_wall = 12 #in
protection = thick_wall/2.0 #6in
len_single_wall = 10*12 #in

area_single_bar = 0.79 #in^2 #8
spacing = 10#in
layer = 2
thick_steel = (1.0*area_single_bar)/spacing*layer #in
h = n_grid_y*len_single_wall + thick_wall #372in
d = h-protection

count = 0
coord_zone = []
for i in range(num_zone):
    coord_zone.append(i*len_single_wall/2.0+protection)


class Concrete_zone:
    def __init__(self, direction, num_ele,top_extend = None ,bot_extend = None , coord =None):
        self.num = num_ele
        self.direction = direction

        if self.direction == 'x':
            self.height = thick_wall
            self.top = coord-thick_wall/2.0
            self.bot = coord+thick_wall/2.0
        if self.direction == 'y':
            if top_extend =='False':
                self.top = coord-(len_single_wall-thick_wall)/2
            if top_extend == 'True':
                self.top = coord-(len_single_wall)/2
            if bot_extend == 'False':
                self.bot = coord +(len_single_wall-thick_wall)/2
            if bot_extend =='True':
                self.bot = coord +(len_single_wall)/2
            self.height = self.bot - self.top

        self.area = self.num*(self.height)*thick_wall
        self.location = (self.top+self.bot)/2.0


class Steel_zone:
    def __init__(self, c,direction, num_ele, location = 0,top = 0,bot = 0,stress = 0,
    ^^Itop_stress = 0, bot_stress = 0):
        self.num_ele = num_ele
        self.direction = direction
        if self.direction == 'x':
            self.location = location
            self.stress = stress
            self.length = len_single_wall
        if self.direction == 'y':
            self.top = top
            self.bot = bot
            self.length = self.bot-self.top
            self.top_stress =top_stress
            self.bot_stress = bot_stress
            self.stress = ( self.top_stress + self.bot_stress)/2.0
            self.location = self.top+(self.top_stress+2.0*self.bot_stress)
            *(self.bot-self.top)/(3.0*(self.top_stress+self.bot_stress))
        self.Pn_i = self.stress*self.num_ele*self.length*thick_steel
        self.Mn_i = self.Pn_i*(center_mass-self.location)
        self.Icr_i = 7*self.num_ele*self.length*thick_steel*(self.location-c)**2.0

# Create an new Excel file and add four worksheet.
workbook_3by3 = xlsxwriter.Workbook('3by3(7)_4sheets_2.xlsx')
worksheet_c_3by3 = workbook_3by3.add_worksheet('c')
worksheet_Pn_3by3 = workbook_3by3.add_worksheet('Pn')
worksheet_Mn_3by3 = workbook_3by3.add_worksheet('Mn')
worksheet_Icr_3by3 = workbook_3by3.add_worksheet('Icr')
```

```
worksheet_c_3by3.write('A1', 'Key')
worksheet_Pn_3by3.write('A1', 'Key')
worksheet_Mn_3by3.write('A1', 'Key')
worksheet_Icr_3by3.write('A1', 'Key')
num_possible_c = (n_grid_y*120+12)/6
for i in range(num_possible_c):

    worksheet_c_3by3.write(0,i+1,'c (in)')
    worksheet_Pn_3by3.write(0,i+1,'Pn (kips)')
    worksheet_Mn_3by3.write(0,i+1,'Mn (kip.ft)')
    worksheet_Icr_3by3.write(0,i+1,'Icr (ft^4)')

list_count_num = []
row = 0
for i_1 in range(n_grid_x+1):
    num_1 = i_1
    for i_2 in range(n_grid_x+1):
        num_2 = i_2+1
        for i_3 in range(n_grid_x+1):
            num_3 = i_3
            for i_4 in range(n_grid_x+1):
                num_4 = i_4+1
                for i_5 in range(n_grid_x+1):
                    num_5 = i_5
                    for i_6 in range(n_grid_x+1):
                        num_6 = i_6+1
                        for i_7 in range(n_grid_x+1):
                            num_7 = i_7

                            count_num = []
                            count_num.append(num_1)
                            count_num.append(num_2)
                            count_num.append(num_3)
                            count_num.append(num_4)
                            count_num.append(num_5)
                            count_num.append(num_6)
                            count_num.append(num_7)
                            row += 1
                            list_count_num.append(count_num)
                            worksheet_c_3by3.write(row,0, str(count_num))
                            worksheet_Pn_3by3.write(row,0, str(count_num))
                            worksheet_Mn_3by3.write(row,0, str(count_num))
                            worksheet_Icr_3by3.write(row,0, str(count_num))

                            list = count_num
                            num_times_coord = 0
                            num_area = 0
                            for i in range(len(count_num)):
                                num_times_coord += count_num[i]*coord_zone[i]
                                num_area += count_num[i]
                            center_mass = num_times_coord/num_area

                            list_Pn = []
                            list_c = []
                            list_Mn = []
                            list_Icr =[]
                            list_all = []

                            for j in range(num_possible_c):
                                c = j*6+3
                                a = 0.8*c

                                #concrete
                                Pn_c = 0
                                Mn_c = 0
                                Icr_c = 0
                                for i in range(len(list)):

                                    if i%2 == 0:
                                        direction = 'x'
            new_concrete_zone = Concrete_zone(direction = 'x',
            ^^Inum_ele = list[i], coord = coord_zone[i])

        if i%2 == 1:
            direction = 'y'
            if list[i-1] == 0:
                if list[i+1]==0:
                    new_concrete_zone = Concrete_zone(direction = 'y',
                    ^^Inum_ele = list[i], top_extend = 'True',
                    ^^Ibot_extend = 'True', coord = coord_zone[i])
                else:
                    new_concrete_zone = Concrete_zone(direction = 'y',
                    ^^Inum_ele = list[i], top_extend = 'True',
                    ^^Ibot_extend = 'False', coord = coord_zone[i])
            if list[i-1] != 0:
                if list[i+1]==0:
                    new_concrete_zone = Concrete_zone(direction = 'y',
                    ^^Inum_ele = list[i], top_extend = 'False',
                    ^^Ibot_extend = 'True', coord = coord_zone[i])
                else:
                    new_concrete_zone = Concrete_zone(direction = 'y',
                    ^^Inum_ele = list[i], top_extend = 'False',
                    ^^Ibot_extend = 'False', coord = coord_zone[i])

        if new_concrete_zone.bot<= a:
            Pn_c += 0.85*concrete_strength*new_concrete_zone.area #kips
            Mn_c += 0.85*concrete_strength*new_concrete_zone.area
            *(center_mass-new_concrete_zone.location)
            Icr_c += new_concrete_zone.area*new_concrete_zone.height
            **2.0/12.0+new_concrete_zone.area*
            (new_concrete_zone.location-c)**2
        else:
            Pn_c += 0.85*concrete_strength*new_concrete_zone.area
            *(a-new_concrete_zone.top)/new_concrete_zone.height
            Mn_c += 0.85*concrete_strength*new_concrete_zone.area
            *(a-new_concrete_zone.top)/new_concrete_zone.height
            *(center_mass-(a+new_concrete_zone.top)/2.0)
            Icr_c += (new_concrete_zone.area/new_concrete_zone.height)
            *((a-new_concrete_zone.top)**3)/12.0+new_concrete_zone.area
            /new_concrete_zone.height*(a-new_concrete_zone.top)
            *(((a+new_concrete_zone.top)/2.0-c)**2.0)

            break


    #steel
    Pn_s = 0
    Mn_s = 0
    Icr_s = 0
    if c <=3*protection:
     ###Top (compression) not yield, Bottom (tension) yield
        list_zone_strain = [c,2/3.0*c,d-5/3.0*c]
        for i in range(len(list)):
            if i%2 == 0:
                direction = 'x'
                location_x = i*len_single_wall/2.0+protection
                if location_x <= (c*5/3.0):
                    stress_x = (c-location_x)/c*0.003/0.002*60
                if location_x >(c*5/3.0):
                    stress_x = -60
                new_steel_zone = Steel_zone(c =c, direction = 'x',
                ^^Inum_ele = list[i], location = location_x,
                ^^Istress = stress_x)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i
            if i%2 == 1:
                direction = 'y'
                top_y = (i-1)*len_single_wall/2.0+protection
                bot_y = (i+1)*len_single_wall/2.0+protection
                if bot_y <= (5/3.0*c):
                    bot_stress_y = (c-bot_y)/c*0.003/0.002*60
                    top_stress_y = (c-top_y)/c*0.003/0.002*60
                    new_steel_zone = Steel_zone(c=c,direction = 'y',
                    ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                    ^^Itop_stress = top_stress_y,
                    ^^Ibot_stress = bot_stress_y)
                    Pn_s += new_steel_zone.Pn_i
                    Mn_s += new_steel_zone.Mn_i
                    Icr_s += new_steel_zone.Icr_i
                if top_y >= (5/3.0*c):
                    bot_stress_y = -60
                    top_stress_y = -60
                    new_steel_zone = Steel_zone(c=c,direction = 'y',
                    ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                    ^^Itop_stress = top_stress_y,
                    ^^Ibot_stress = bot_stress_y)
                    Pn_s += new_steel_zone.Pn_i
                    Mn_s += new_steel_zone.Mn_i
                    Icr_s += new_steel_zone.Icr_i

                if top_y < (5/3.0*c) and bot_y >(5/3.0*c):
                    top1_y = top_y
                    bot1_y = 5/3.0*c
                    top1_stress_y = (c-top1_y)/c*0.003/0.002*60
                    bot1_stress_y = (c-bot1_y)/c*0.003/0.002*60
                    new_steel_zone1 = Steel_zone(c=c,direction = 'y',
                    ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
                    ^^Itop_stress = top1_stress_y,
                    ^^Ibot_stress = bot1_stress_y)
                    Pn_s += new_steel_zone1.Pn_i
                    Mn_s += new_steel_zone1.Mn_i
                    Icr_s += new_steel_zone.Icr_i
                    top2_y = 5/3.0*c
                    bot2_y = bot_y
                    top2_stress_y = -60
                    bot2_stress_y = -60
                    new_steel_zone2 = Steel_zone(c=c,direction = 'y',
                    ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
                    ^^Itop_stress = top2_stress_y,
                    ^^Ibot_stress = bot2_stress_y)
                    Pn_s += new_steel_zone2.Pn_i
                    Mn_s += new_steel_zone2.Mn_i
                    Icr_s += new_steel_zone.Icr_i

    if c > 3*protection and c <= 3/5.0*d:
    ###Top (compression) yield, Bottom (tension) yield
        list_zone_strain = [1/3*c,2/3*c,2/3*c,d-5/3*c]
        for i in range(len(list)):
            if i%2 == 0:
                direction = 'x'
                location_x = i*len_single_wall/2.0+protection
                if location_x <= (1/3.0*c):
                    stress_x = 60
                if location_x > (1/3.0*c) and location_x <= (c*5/3.0):
                    stress_x = (c-location_x)/c*0.003/0.002*60
                if location_x >(c*5/3.0):
                    stress_x = -60
                new_steel_zone = Steel_zone(c=c,direction = 'x',
                ^^Inum_ele = list[i], location = location_x,
                ^^Istress = stress_x)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i
            if i%2 == 1:
                direction = 'y'
                top_y = (i-1)*len_single_wall/2.0+protection
                bot_y = (i+1)*len_single_wall/2.0+protection
                if bot_y <= (1/3.0*c):
                    bot_stress_y = 60
                    top_stress_y = 60
                    new_steel_zone= Steel_zone(c=c,direction = 'y',
        ^^Inum_ele = list[i], top = top_y,bot = bot_y,
        ^^Itop_stress = top_stress_y,
        ^^Ibot_stress = bot_stress_y)
        Pn_s += new_steel_zone1.Pn_i
        Mn_s += new_steel_zone1.Mn_i
        Icr_s += new_steel_zone.Icr_i
    if bot_y > (1/3.0*c) and top_y <= (1/3.0*c):
        top1_y = top_y
        bot1_y = 1/3.0*c
        top1_stress_y = 60
        bot1_stress_y = 60
        new_steel_zone1 = Steel_zone(c=c,direction = 'y',
        ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
        ^^Itop_stress = top1_stress_y,
        ^^Ibot_stress = bot1_stress_y)
        Pn_s += new_steel_zone1.Pn_i
        Mn_s += new_steel_zone1.Mn_i
        Icr_s += new_steel_zone.Icr_i
        top2_y = 1/3.0*c
        bot2_y = bot_y
        top1_stress_y = (c-top2_y)/c*0.003/0.002*60
        bot1_stress_y = (c-bot2_y)/c*0.003/0.002*60
        new_steel_zone2 = Steel_zone(c=c,direction = 'y',
        ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
        ^^Itop_stress = top2_stress_y,
        ^^Ibot_stress = bot2_stress_y)
        Pn_s += new_steel_zone2.Pn_i
        Mn_s += new_steel_zone2.Mn_i
        Icr_s += new_steel_zone.Icr_i
    if bot_y <= (5/3.0*c) and top_y > (1/3.0*c):
        bot_stress_y = (c-bot_y)/c*0.003/0.002*60
        top_stress_y = (c-top_y)/c*0.003/0.002*60
        new_steel_zone = Steel_zone(c=c,direction = 'y',
        ^^Inum_ele = list[i], top = top_y,bot = bot_y,
        ^^Itop_stress = top_stress_y,
        ^^Ibot_stress = bot_stress_y)
        Pn_s += new_steel_zone.Pn_i
        Mn_s += new_steel_zone.Mn_i
        Icr_s += new_steel_zone.Icr_i
    if top_y > (5/3.0*c):
        bot_stress_y = -60
        top_stress_y = -60
        new_steel_zone = Steel_zone(c=c,direction = 'y',
        ^^Inum_ele = list[i], top = top_y,bot = bot_y,
        ^^Itop_stress = top_stress_y,
        ^^Ibot_stress = bot_stress_y)
        Pn_s += new_steel_zone.Pn_i
        Mn_s += new_steel_zone.Mn_i
        Icr_s += new_steel_zone.Icr_i

    if top_y <= (5/3.0*c) and bot_y >(5/3.0*c):
        top1_y = top_y
        bot1_y = 5/3.0*c
        top1_stress_y = (c-top1_y)/c*0.003/0.002*60
        bot1_stress_y = (c-bot1_y)/c*0.003/0.002*60
        new_steel_zone1 = Steel_zone(c=c,direction = 'y',
        ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
        ^^Itop_stress = top1_stress_y,
        ^^Ibot_stress = bot1_stress_y)
        Pn_s += new_steel_zone1.Pn_i
        Mn_s += new_steel_zone1.Mn_i
        Icr_s += new_steel_zone.Icr_i
        top2_y = 5/3.0*c
        bot2_y = bot_y
        top2_stress_y = -60
        bot2_stress_y = -60
        new_steel_zone2 = Steel_zone(c=c,direction = 'y',
        ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
        ^^Itop_stress = top2_stress_y,
        ^^Ibot_stress = bot2_stress_y)
        Pn_s += new_steel_zone2.Pn_i
```

```
                Mn_s += new_steel_zone2.Mn_i
                Icr_s += new_steel_zone.Icr_i
if c > (3.0/5*d):
    list_zone_strain = [1/3*c,2/3.0*c,d-c]
    for i in range(len(list)):
        if i%2 == 0:
            direction = 'x'
            location_x = i*len_single_wall/2.0+protection
            if location_x <= (c*1/3.0):
                stress_x = 60
            if location_x > (c*1/3.0):
                stress_x = (c-location_x)/c*0.003/0.002*60
            new_steel_zone = Steel_zone(c=c,direction = 'x',
            ^^Inum_ele = list[i], location = location_x,
            ^^Istress = stress_x)
            Pn_s += new_steel_zone.Pn_i
            Mn_s += new_steel_zone.Mn_i
            Icr_s += new_steel_zone.Icr_i
        if i%2 == 1:
            direction = 'y'
            top_y = (i-1)*len_single_wall/2.0+protection
            bot_y = (i+1)*len_single_wall/2.0+protection
            if bot_y <= (1/3.0*c):
                bot_stress_y = 60
                top_stress_y = 60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y,
                ^^Ibot_stress = bot_stress_y)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i
            if top_y >= (1/3.0*c):
                bot_stress_y = (c-bot_y)/c*0.003/0.002*60
                top_stress_y = (c-top_y)/c*0.003/0.002*60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y,
                ^^Ibot_stress = bot_stress_y)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i

            if top_y < (1/3.0*c) and bot_y >(1/3.0*c):
                top1_y = top_y
                bot1_y = 1/3.0*c
                top1_stress_y = 60
                bot1_stress_y = 60
                new_steel_zone1 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
                ^^Itop_stress = top1_stress_y,
                ^^Ibot_stress = bot1_stress_y)
                Pn_s += new_steel_zone1.Pn_i
                Mn_s += new_steel_zone1.Mn_i
                Icr_s += new_steel_zone.Icr_i
                top2_y = 1/3.0*c
                bot2_y = bot_y
                top2_stress_y = (c-top2_y)/c*0.003/0.002*60
                bot2_stress_y = (c-bot2_y)/c*0.003/0.002*60
                new_steel_zone2 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
                ^^Itop_stress = top2_stress_y,
                ^^Ibot_stress = bot2_stress_y)
                Pn_s += new_steel_zone2.Pn_i
                Mn_s += new_steel_zone2.Mn_i
                Icr_s += new_steel_zone.Icr_i

Pn = round(0.9*(Pn_s+ Pn_c))
Mn = round(0.9*(Mn_s + Mn_c)/12)
Icr =  (Icr_s+ Icr_c)/(12**4)
```

```
                                list_Mn.append(Mn)
                                list_Pn.append(Pn)
                                list_Icr.append(Icr)
                                list_c.append(c)
                                list_all.append(c)
                                list_all.append(Pn)
                                list_all.append(Mn)
                                list_all.append(Icr)

                            dict_all[str(count_num)]=[list_all]

                            for i in range(len(list_c)):
                                worksheet_c_3by3.write(row,i+1,list_c[i])
                                worksheet_Pn_3by3.write(row,i+1,list_Pn[i])
                                worksheet_Mn_3by3.write(row,i+1,list_Mn[i])
                                worksheet_Icr_3by3.write(row,i+1,list_Icr[i])

workbook_3by3.close()

######3by2
###algorithm parameters
n_grid_x = 3
n_grid_y = 2 #grid size: n_grid_x by n_grid_y
num_zone = n_grid_y*2+1

h = n_grid_y*len_single_wall + thick_wall #372in
d = h-protection

coord_zone = []
for i in range(num_zone):
    coord_zone.append(i*len_single_wall/2.0+protection)

# Create an new Excel file and add four worksheet.
workbook_3by2 = xlsxwriter.Workbook('3by2(5)_4sheets_2.xlsx')
worksheet_c_3by2 = workbook_3by2.add_worksheet('c')
worksheet_Pn_3by2 = workbook_3by2.add_worksheet('Pn')
worksheet_Mn_3by2 = workbook_3by2.add_worksheet('Mn')
worksheet_Icr_3by2 = workbook_3by2.add_worksheet('Icr')


worksheet_c_3by2.write('A1', 'Key')
worksheet_Pn_3by2.write('A1', 'Key')
worksheet_Mn_3by2.write('A1', 'Key')
worksheet_Icr_3by2.write('A1', 'Key')
num_possible_c = (n_grid_y*120+12)/6

for i in range(num_possible_c):
    worksheet_c_3by2.write(0,i+1,'c (in)')
    worksheet_Pn_3by2.write(0,i+1,'Pn (kips)')
    worksheet_Mn_3by2.write(0,i+1,'Mn (kip.ft)')
    worksheet_Icr_3by2.write(0,i+1,'Icr (ft^4)')

list_count_num = []
row = 0
for i_1 in range(n_grid_x+1):
    num_1 = i_1
    for i_2 in range(n_grid_x+1):
        num_2 = i_2+1
        for i_3 in range(n_grid_x+1):
            num_3 = i_3
            for i_4 in range(n_grid_x+1):
                num_4 = i_4+1
                for i_5 in range(n_grid_x+1):
                    num_5 = i_5

                    count_num = []
                    count_num.append(num_1)
                    count_num.append(num_2)
                    count_num.append(num_3)
                    count_num.append(num_4)
                    count_num.append(num_5)
```

```
row += 1
list_count_num.append(count_num)
worksheet_c_3by2.write(row,0, str(count_num))
worksheet_Pn_3by2.write(row,0, str(count_num))
worksheet_Mn_3by2.write(row,0, str(count_num))
worksheet_Icr_3by2.write(row,0, str(count_num))

list = count_num
num_times_coord = 0
num_area = 0

for i in range(len(count_num)):
    num_times_coord += count_num[i]*coord_zone[i]
    num_area += count_num[i]
center_mass = num_times_coord/num_area

list_Pn = []
list_c = []
list_Mn = []
list_Icr =[]
list_all = []

for j in range(num_possible_c):
    c = j*6+3
    a = 0.8*c

    #concrete
    Pn_c = 0
    Mn_c = 0
    Icr_c = 0
    for i in range(len(list)):

        if i%2 == 0:
            direction = 'x'
            new_concrete_zone = Concrete_zone(direction = 'x',
            ^^Inum_ele = list[i], coord = coord_zone[i])

        if i%2 == 1:
            direction = 'y'
            if list[i-1] == 0:
                if list[i+1]==0:
                    new_concrete_zone = Concrete_zone(direction = 'y',
                    ^^Inum_ele = list[i], top_extend = 'True',
                    ^^Ibot_extend = 'True', coord = coord_zone[i])
                else:
                    new_concrete_zone = Concrete_zone(direction = 'y',
                    ^^Inum_ele = list[i], top_extend = 'True',
                    ^^Ibot_extend = 'False', coord = coord_zone[i])
            if list[i-1] != 0:
                if list[i+1]==0:
                    new_concrete_zone = Concrete_zone(direction = 'y',
                    ^^Inum_ele = list[i], top_extend = 'False',
                    ^^Ibot_extend = 'True', coord = coord_zone[i])
                else:
                    new_concrete_zone = Concrete_zone(direction = 'y',
                    ^^Inum_ele = list[i], top_extend = 'False',
                    ^^Ibot_extend = 'False', coord = coord_zone[i])

        if new_concrete_zone.bot<= a:
            Pn_c += 0.85*concrete_strength*new_concrete_zone.area #kips
            Mn_c += 0.85*concrete_strength*new_concrete_zone.area
            *(center_mass-new_concrete_zone.location)
            Icr_c += new_concrete_zone.area*new_concrete_zone.height**2.0
            /12.0+new_concrete_zone.area*(new_concrete_zone.location-c)**2
        else:
            Pn_c += 0.85*concrete_strength*new_concrete_zone.area
            *(a-new_concrete_zone.top)/new_concrete_zone.height
            Mn_c += 0.85*concrete_strength*new_concrete_zone.area
            *(a-new_concrete_zone.top)/new_concrete_zone.height
            *(center_mass-(a+new_concrete_zone.top)/2.0)
        Icr_c += (new_concrete_zone.area/new_concrete_zone.height)
        *((a-new_concrete_zone.top)**3)/12.0+new_concrete_zone.area
        /new_concrete_zone.height*(a-new_concrete_zone.top)
        *(((a+new_concrete_zone.top)/2.0-c)**2.0)
        break


#steel
Pn_s = 0
Mn_s = 0
Icr_s = 0
if c <=3*protection:
###Top (compression) not yield, Bottom (tension) yield
    list_zone_strain = [c,2/3.0*c,d-5/3.0*c]
    for i in range(len(list)):
        if i%2 == 0:
            direction = 'x'
            location_x = i*len_single_wall/2.0+protection
            if location_x <= (c*5/3.0):
                stress_x = (c-location_x)/c*0.003/0.002*60
            if location_x >(c*5/3.0):
                stress_x = -60
            new_steel_zone = Steel_zone(c =c, direction = 'x',
            ^^Inum_ele = list[i], location = location_x,stress = stress_x)
            Pn_s += new_steel_zone.Pn_i
            Mn_s += new_steel_zone.Mn_i
            Icr_s += new_steel_zone.Icr_i
        if i%2 == 1:
            direction = 'y'
            top_y = (i-1)*len_single_wall/2.0+protection
            bot_y = (i+1)*len_single_wall/2.0+protection
            if bot_y <= (5/3.0*c):
                bot_stress_y = (c-bot_y)/c*0.003/0.002*60
                top_stress_y = (c-top_y)/c*0.003/0.002*60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y, bot_stress = bot_stress_y)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i
            if top_y >= (5/3.0*c):
                bot_stress_y = -60
                top_stress_y = -60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y, bot_stress = bot_stress_y)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i

            if top_y < (5/3.0*c) and bot_y >(5/3.0*c):
                top1_y = top_y
                bot1_y = 5/3.0*c
                top1_stress_y = (c-top1_y)/c*0.003/0.002*60
                bot1_stress_y = (c-bot1_y)/c*0.003/0.002*60
                new_steel_zone1 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
                ^^Itop_stress = top1_stress_y,
                ^^Ibot_stress = bot1_stress_y)
                Pn_s += new_steel_zone1.Pn_i
                Mn_s += new_steel_zone1.Mn_i
                Icr_s += new_steel_zone.Icr_i
                top2_y = 5/3.0*c
                bot2_y = bot_y
                top2_stress_y = -60
                bot2_stress_y = -60
                new_steel_zone2 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
                ^^Itop_stress = top2_stress_y,
                ^^Ibot_stress = bot2_stress_y)
                Pn_s += new_steel_zone2.Pn_i
                Mn_s += new_steel_zone2.Mn_i
                Icr_s += new_steel_zone.Icr_i


if c > 3*protection and c <= 3/5.0*d:
###Top (compression) yield, Bottom (tension) yield
    list_zone_strain = [1/3*c,2/3*c,2/3*c,d-5/3*c]
    for i in range(len(list)):
        if i%2 == 0:
            direction = 'x'
            location_x = i*len_single_wall/2.0+protection
            if location_x <= (1/3.0*c):
                stress_x = 60
            if location_x > (1/3.0*c) and location_x <= (c*5/3.0):
                stress_x = (c-location_x)/c*0.003/0.002*60
            if location_x >(c*5/3.0):
                stress_x = -60
            new_steel_zone = Steel_zone(c=c,direction = 'x',
            ^^Inum_ele = list[i], location = location_x,stress = stress_x)
            Pn_s += new_steel_zone.Pn_i
            Mn_s += new_steel_zone.Mn_i
            Icr_s += new_steel_zone.Icr_i
        if i%2 == 1:
            direction = 'y'
            top_y = (i-1)*len_single_wall/2.0+protection
            bot_y = (i+1)*len_single_wall/2.0+protection
            if bot_y <= (1/3.0*c):
                bot_stress_y = 60
                top_stress_y = 60
                new_steel_zone= Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y,
                ^^Ibot_stress = bot_stress_y)
                Pn_s += new_steel_zone1.Pn_i
                Mn_s += new_steel_zone1.Mn_i
                Icr_s += new_steel_zone.Icr_i
            if bot_y > (1/3.0*c) and top_y <= (1/3.0*c):
                top1_y = top_y
                bot1_y = 1/3.0*c
                top1_stress_y = 60
                bot1_stress_y = 60
                new_steel_zone1 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
                ^^Itop_stress = top1_stress_y,
                ^^Ibot_stress = bot1_stress_y)
                Pn_s += new_steel_zone1.Pn_i
                Mn_s += new_steel_zone1.Mn_i
                Icr_s += new_steel_zone.Icr_i
                top2_y = 1/3.0*c
                bot2_y = bot_y
                top1_stress_y = (c-top2_y)/c*0.003/0.002*60
                bot1_stress_y = (c-bot2_y)/c*0.003/0.002*60
                new_steel_zone2 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
                ^^Itop_stress = top2_stress_y,
                ^^Ibot_stress = bot2_stress_y)
                Pn_s += new_steel_zone2.Pn_i
                Mn_s += new_steel_zone2.Mn_i
                Icr_s += new_steel_zone.Icr_i
            if bot_y <= (5/3.0*c) and top_y > (1/3.0*c):
                bot_stress_y = (c-bot_y)/c*0.003/0.002*60
                top_stress_y = (c-top_y)/c*0.003/0.002*60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y,
                ^^Ibot_stress = bot_stress_y)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i
            if top_y > (5/3.0*c):
                bot_stress_y = -60
                top_stress_y = -60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y,
                ^^Ibot_stress = bot_stress_y)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i

            if top_y <= (5/3.0*c) and bot_y >(5/3.0*c):
                top1_y = top_y
                bot1_y = 5/3.0*c
                top1_stress_y = (c-top1_y)/c*0.003/0.002*60
                bot1_stress_y = (c-bot1_y)/c*0.003/0.002*60
                new_steel_zone1 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
                ^^Itop_stress = top1_stress_y,
                ^^Ibot_stress = bot1_stress_y)
                Pn_s += new_steel_zone1.Pn_i
                Mn_s += new_steel_zone1.Mn_i
                Icr_s += new_steel_zone.Icr_i
                top2_y = 5/3.0*c
                bot2_y = bot_y
                top2_stress_y = -60
                bot2_stress_y = -60
                new_steel_zone2 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
                ^^Itop_stress = top2_stress_y,
                ^^Ibot_stress = bot2_stress_y)
                Pn_s += new_steel_zone2.Pn_i
                Mn_s += new_steel_zone2.Mn_i
                Icr_s += new_steel_zone.Icr_i
if c > (3.0/5*d):

    #break

    list_zone_strain = [1/3*c,2/3.0*c,d-c]
    for i in range(len(list)):
        if i%2 == 0:
            direction = 'x'
            location_x = i*len_single_wall/2.0+protection
            if location_x <= (c*1/3.0):
                stress_x = 60
            if location_x > (c*1/3.0):
                stress_x = (c-location_x)/c*0.003/0.002*60
            new_steel_zone = Steel_zone(c=c,direction = 'x',
            ^^Inum_ele = list[i], location = location_x,
            ^^Istress = stress_x)
            Pn_s += new_steel_zone.Pn_i
            Mn_s += new_steel_zone.Mn_i
            Icr_s += new_steel_zone.Icr_i
        if i%2 == 1:
            direction = 'y'
            top_y = (i-1)*len_single_wall/2.0+protection
            bot_y = (i+1)*len_single_wall/2.0+protection
            if bot_y <= (1/3.0*c):
                bot_stress_y = 60
                top_stress_y = 60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y,
                ^^Ibot_stress = bot_stress_y)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i
            if top_y >= (1/3.0*c):
                bot_stress_y = (c-bot_y)/c*0.003/0.002*60
                top_stress_y = (c-top_y)/c*0.003/0.002*60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y,
                                        ^^Ibot_stress = bot_stress_y)
                                        Pn_s += new_steel_zone.Pn_i
                                        Mn_s += new_steel_zone.Mn_i
                                        Icr_s += new_steel_zone.Icr_i

                                    if top_y < (1/3.0*c) and bot_y >(1/3.0*c):
                                        top1_y = top_y
                                        bot1_y = 1/3.0*c
                                        top1_stress_y = 60
                                        bot1_stress_y = 60
                                        new_steel_zone1 = Steel_zone(c=c,direction = 'y',
                                        ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
                                        ^^Itop_stress = top1_stress_y,
                                        ^^Ibot_stress = bot1_stress_y)
                                        Pn_s += new_steel_zone1.Pn_i
                                        Mn_s += new_steel_zone1.Mn_i
                                        Icr_s += new_steel_zone.Icr_i
                                        top2_y = 1/3.0*c
                                        bot2_y = bot_y
                                        top2_stress_y = (c-top2_y)/c*0.003/0.002*60
                                        bot2_stress_y = (c-bot2_y)/c*0.003/0.002*60
                                        new_steel_zone2 = Steel_zone(c=c,direction = 'y',
                                        ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
                                        ^^Itop_stress = top2_stress_y,
                                        ^^Ibot_stress = bot2_stress_y)
                                        Pn_s += new_steel_zone2.Pn_i
                                        Mn_s += new_steel_zone2.Mn_i
                                        Icr_s += new_steel_zone.Icr_i


                        Pn = round(0.9*(Pn_s+ Pn_c))
                        Mn = round(0.9*(Mn_s + Mn_c)/12)
                        Icr =  (Icr_s+ Icr_c)/(12**4)

                        list_Mn.append(Mn)
                        list_Pn.append(Pn)
                        list_Icr.append(Icr)
                        list_c.append(c)
                        list_all.append(c)
                        list_all.append(Pn)
                        list_all.append(Mn)
                        list_all.append(Icr)

                    for i in range(len(list_c)):
                        worksheet_c_3by2.write(row,i+1,list_c[i])
                        worksheet_Pn_3by2.write(row,i+1,list_Pn[i])
                        worksheet_Mn_3by2.write(row,i+1,list_Mn[i])
                        worksheet_Icr_3by2.write(row,i+1,list_Icr[i])
                    dict_all[str(count_num)]=[list_all]
workbook_3by2.close()

######3by1
###algorithm parameters
n_grid_x = 3
n_grid_y = 1 #grid size: n_grid_x by n_grid_y
num_zone = n_grid_y*2+1

h = n_grid_y*len_single_wall + thick_wall #372in
d = h-protection

coord_zone = []
for i in range(num_zone):
    coord_zone.append(i*len_single_wall/2.0+protection)

# Create an new Excel file and add four worksheet.
workbook_3by1 = xlsxwriter.Workbook('3by1(3)_4sheets_2.xlsx')
worksheet_c_3by1 = workbook_3by1.add_worksheet('c')
worksheet_Pn_3by1 = workbook_3by1.add_worksheet('Pn')
worksheet_Mn_3by1 = workbook_3by1.add_worksheet('Mn')
worksheet_Icr_3by1 = workbook_3by1.add_worksheet('Icr')
```

```
worksheet_c_3by1.write('A1', 'Key')
worksheet_Pn_3by1.write('A1', 'Key')
worksheet_Mn_3by1.write('A1', 'Key')
worksheet_Icr_3by1.write('A1', 'Key')
num_possible_c = (n_grid_y*120+12)/6

for i in range(num_possible_c):
    worksheet_c_3by1.write(0,i+1,'c (in)')
    worksheet_Pn_3by1.write(0,i+1,'Pn (kips)')
    worksheet_Mn_3by1.write(0,i+1,'Mn (kip.ft)')
    worksheet_Icr_3by1.write(0,i+1,'Icr (ft^4)')

list_count_num = []
row = 0
for i_1 in range(n_grid_x+1):
    num_1 = i_1
    for i_2 in range(n_grid_x+1):
        num_2 = i_2+1
        for i_3 in range(n_grid_x+1):
            num_3 = i_3
            count_num = []
            count_num.append(num_1)
            count_num.append(num_2)
            count_num.append(num_3)

            row += 1
            list_count_num.append(count_num)
            worksheet_c_3by1.write(row,0, str(count_num))
            worksheet_Pn_3by1.write(row,0, str(count_num))
            worksheet_Mn_3by1.write(row,0, str(count_num))
            worksheet_Icr_3by1.write(row,0, str(count_num))

            list = count_num
            num_times_coord = 0
            num_area = 0

            for i in range(len(count_num)):
                num_times_coord += count_num[i]*coord_zone[i]
                num_area += count_num[i]
            center_mass = num_times_coord/num_area

            list_Pn = []
            list_c = []
            list_Mn = []
            list_Icr =[]
            list_all = []
            for j in range(num_possible_c):
                c = j*6+3
                a = 0.8*c

                #concrete
                Pn_c = 0
                Mn_c = 0
                Icr_c = 0
                for i in range(len(list)):

                    if i%2 == 0:
                        direction = 'x'
                        new_concrete_zone = Concrete_zone(direction = 'x',
                        ^^Inum_ele = list[i], coord = coord_zone[i])

                    if i%2 == 1:
                        direction = 'y'
                        if list[i-1] == 0:
                            if list[i+1]==0:
                                new_concrete_zone = Concrete_zone(direction = 'y',
                                ^^Inum_ele = list[i], top_extend = 'True',
                                ^^Ibot_extend = 'True', coord = coord_zone[i])
                            else:
                                new_concrete_zone = Concrete_zone(direction = 'y',
                ^^Inum_ele = list[i], top_extend = 'True',
                ^^Ibot_extend = 'False', coord = coord_zone[i])
        if list[i-1] != 0:
            if list[i+1]==0:
                new_concrete_zone = Concrete_zone(direction = 'y',
                ^^Inum_ele = list[i], top_extend = 'False',
                ^^Ibot_extend = 'True', coord = coord_zone[i])
            else:
                new_concrete_zone = Concrete_zone(direction = 'y',
                ^^Inum_ele = list[i], top_extend = 'False',
                ^^Ibot_extend = 'False', coord = coord_zone[i])

    if new_concrete_zone.bot<= a:
        Pn_c += 0.85*concrete_strength*new_concrete_zone.area #kips
        Mn_c += 0.85*concrete_strength*new_concrete_zone.area
        *(center_mass-new_concrete_zone.location)
        Icr_c += new_concrete_zone.area*new_concrete_zone.height**2.0
        /12.0+new_concrete_zone.area*(new_concrete_zone.location-c)**2
    else:
        Pn_c += 0.85*concrete_strength*new_concrete_zone.area
        *(a-new_concrete_zone.top)/new_concrete_zone.height
        Mn_c += 0.85*concrete_strength*new_concrete_zone.area
        *(a-new_concrete_zone.top)/new_concrete_zone.height
        *(center_mass-(a+new_concrete_zone.top)/2.0)
        Icr_c += (new_concrete_zone.area/new_concrete_zone.height)
        *((a-new_concrete_zone.top)**3)/12.0+new_concrete_zone.area
        /new_concrete_zone.height*(a-new_concrete_zone.top)
        *(((a+new_concrete_zone.top)/2.0-c)**2.0)
        break

#steel
Pn_s = 0
Mn_s = 0
Icr_s = 0
if c <=3*protection: ###Top (compression) not yield, Bottom (tension) yield
    list_zone_strain = [c,2/3.0*c,d-5/3.0*c]
    for i in range(len(list)):
        if i%2 == 0:
            direction = 'x'
            location_x = i*len_single_wall/2.0+protection
            if location_x <= (c*5/3.0):
                stress_x = (c-location_x)/c*0.003/0.002*60
            if location_x >(c*5/3.0):
                stress_x = -60
            new_steel_zone = Steel_zone(c =c, direction = 'x',
            ^^Inum_ele = list[i], location = location_x,
            ^^Istress = stress_x)
            Pn_s += new_steel_zone.Pn_i
            Mn_s += new_steel_zone.Mn_i
            Icr_s += new_steel_zone.Icr_i
        if i%2 == 1:
            direction = 'y'
            top_y = (i-1)*len_single_wall/2.0+protection
            bot_y = (i+1)*len_single_wall/2.0+protection
            if bot_y <= (5/3.0*c):
                bot_stress_y = (c-bot_y)/c*0.003/0.002*60
                top_stress_y = (c-top_y)/c*0.003/0.002*60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y, bot_stress = bot_stress_y)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i
            if top_y >= (5/3.0*c):
                bot_stress_y = -60
                top_stress_y = -60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y, bot_stress = bot_stress_y)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += new_steel_zone.Icr_i

            if top_y < (5/3.0*c) and bot_y >(5/3.0*c):
                top1_y = top_y
                bot1_y = 5/3.0*c
                top1_stress_y = (c-top1_y)/c*0.003/0.002*60
                bot1_stress_y = (c-bot1_y)/c*0.003/0.002*60
                new_steel_zone1 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i],top = top1_y,bot = bot1_y,
                ^^Itop_stress = top1_stress_y, bot_stress = bot1_stress_y)
                Pn_s += new_steel_zone1.Pn_i
                Mn_s += new_steel_zone1.Mn_i
                Icr_s += new_steel_zone.Icr_i
                top2_y = 5/3.0*c
                bot2_y = bot_y
                top2_stress_y = -60
                bot2_stress_y = -60
                new_steel_zone2 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
                ^^Itop_stress = top2_stress_y, bot_stress = bot2_stress_y)
                Pn_s += new_steel_zone2.Pn_i
                Mn_s += new_steel_zone2.Mn_i
                Icr_s += new_steel_zone.Icr_i

if c > 3*protection and c <= 3/5.0*d:
###Top (compression) yield, Bottom (tension) yield
    list_zone_strain = [1/3*c,2/3*c,2/3*c,d-5/3*c]
    for i in range(len(list)):
        if i%2 == 0:
            direction = 'x'
            location_x = i*len_single_wall/2.0+protection
            if location_x <= (1/3.0*c):
                stress_x = 60
            if location_x > (1/3.0*c) and location_x <= (c*5/3.0):
                stress_x = (c-location_x)/c*0.003/0.002*60
            if location_x >(c*5/3.0):
                stress_x = -60
            new_steel_zone = Steel_zone(c=c,direction = 'x', num_ele = list[i],
            ^^Ilocation = location_x,stress = stress_x)
            Pn_s += new_steel_zone.Pn_i
            Mn_s += new_steel_zone.Mn_i
            Icr_s += new_steel_zone.Icr_i
        if i%2 == 1:
            direction = 'y'
            top_y = (i-1)*len_single_wall/2.0+protection
            bot_y = (i+1)*len_single_wall/2.0+protection
            if bot_y <= (1/3.0*c):
                bot_stress_y = 60
                top_stress_y = 60
                new_steel_zone= Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y, bot_stress = bot_stress_y)
                Pn_s += new_steel_zone1.Pn_i
                Mn_s += new_steel_zone1.Mn_i
                Icr_s += new_steel_zone.Icr_i
            if bot_y > (1/3.0*c) and top_y <= (1/3.0*c):
                top1_y = top_y
                bot1_y = 1/3.0*c
                top1_stress_y = 60
                bot1_stress_y = 60
                new_steel_zone1 = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
                ^^Itop_stress = top1_stress_y, bot_stress = bot1_stress_y)
                Pn_s += new_steel_zone1.Pn_i
                Mn_s += new_steel_zone1.Mn_i
                Icr_s += new_steel_zone.Icr_i
                top2_y = 1/3.0*c
                bot2_y = bot_y
                top1_stress_y = (c-top2_y)/c*0.003/0.002*60
                bot1_stress_y = (c-bot2_y)/c*0.003/0.002*60
                new_steel_zone2 = Steel_zone(c=c,direction = 'y',
            ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
            ^^Itop_stress = top2_stress_y, bot_stress = bot2_stress_y)
            Pn_s += new_steel_zone2.Pn_i
            Mn_s += new_steel_zone2.Mn_i
            Icr_s += new_steel_zone.Icr_i
        if bot_y <= (5/3.0*c) and top_y > (1/3.0*c):
            bot_stress_y = (c-bot_y)/c*0.003/0.002*60
            top_stress_y = (c-top_y)/c*0.003/0.002*60
            new_steel_zone = Steel_zone(c=c,direction = 'y',
            ^^Inum_ele = list[i], top = top_y,bot = bot_y,
            ^^Itop_stress = top_stress_y, bot_stress = bot_stress_y)
            Pn_s += new_steel_zone.Pn_i
            Mn_s += new_steel_zone.Mn_i
            Icr_s += new_steel_zone.Icr_i
        if top_y > (5/3.0*c):
            bot_stress_y = -60
            top_stress_y = -60
            new_steel_zone = Steel_zone(c=c,direction = 'y',
            ^^Inum_ele = list[i], top = top_y,bot = bot_y,
            ^^Itop_stress = top_stress_y, bot_stress = bot_stress_y)
            Pn_s += new_steel_zone.Pn_i
            Mn_s += new_steel_zone.Mn_i
            Icr_s += new_steel_zone.Icr_i

        if top_y <= (5/3.0*c) and bot_y >(5/3.0*c):
            top1_y = top_y
            bot1_y = 5/3.0*c
            top1_stress_y = (c-top1_y)/c*0.003/0.002*60
            bot1_stress_y = (c-bot1_y)/c*0.003/0.002*60
            new_steel_zone1 = Steel_zone(c=c,direction = 'y',
            ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
            ^^Itop_stress = top1_stress_y, bot_stress = bot1_stress_y)
            Pn_s += new_steel_zone1.Pn_i
            Mn_s += new_steel_zone1.Mn_i
            Icr_s += new_steel_zone.Icr_i
            top2_y = 5/3.0*c
            bot2_y = bot_y
            top2_stress_y = -60
            bot2_stress_y = -60
            new_steel_zone2 = Steel_zone(c=c,direction = 'y',
            ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
            ^^Itop_stress = top2_stress_y, bot_stress = bot2_stress_y)
            Pn_s += new_steel_zone2.Pn_i
            Mn_s += new_steel_zone2.Mn_i
            Icr_s += new_steel_zone.Icr_i
if c > (3.0/5*d):

    list_zone_strain = [1/3*c,2/3.0*c,d-c]
    for i in range(len(list)):
        if i%2 == 0:
            direction = 'x'
            location_x = i*len_single_wall/2.0+protection
            if location_x <= (c*1/3.0):
                stress_x = 60
            if location_x > (c*1/3.0):
                stress_x = (c-location_x)/c*0.003/0.002*60
            new_steel_zone = Steel_zone(c=c,direction = 'x', num_ele = list[i],
            ^^Ilocation = location_x,stress = stress_x)
            Pn_s += new_steel_zone.Pn_i
            Mn_s += new_steel_zone.Mn_i
            Icr_s += new_steel_zone.Icr_i
        if i%2 == 1:
            direction = 'y'
            top_y = (i-1)*len_single_wall/2.0+protection
            bot_y = (i+1)*len_single_wall/2.0+protection
            if bot_y <= (1/3.0*c):
                bot_stress_y = 60
                top_stress_y = 60
                new_steel_zone = Steel_zone(c=c,direction = 'y',
                ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                ^^Itop_stress = top_stress_y, bot_stress = bot_stress_y)
```

```
                                        Pn_s += new_steel_zone.Pn_i
                                        Mn_s += new_steel_zone.Mn_i
                                        Icr_s += new_steel_zone.Icr_i
                                    if top_y >= (1/3.0*c):
                                        bot_stress_y = (c-bot_y)/c*0.003/0.002*60
                                        top_stress_y = (c-top_y)/c*0.003/0.002*60
                                        new_steel_zone = Steel_zone(c=c,direction = 'y',
                                        ^^Inum_ele = list[i], top = top_y,bot = bot_y,
                                        ^^Itop_stress = top_stress_y, bot_stress = bot_stress_y)
                                        Pn_s += new_steel_zone.Pn_i
                                        Mn_s += new_steel_zone.Mn_i
                                        Icr_s += new_steel_zone.Icr_i

                                    if top_y < (1/3.0*c) and bot_y >(1/3.0*c):
                                        top1_y = top_y
                                        bot1_y = 1/3.0*c
                                        top1_stress_y = 60
                                        bot1_stress_y = 60
                                        new_steel_zone1 = Steel_zone(c=c,direction = 'y',
                                        ^^Inum_ele = list[i], top = top1_y,bot = bot1_y,
                                        ^^Itop_stress = top1_stress_y, bot_stress = bot1_stress_y)
                                        Pn_s += new_steel_zone1.Pn_i
                                        Mn_s += new_steel_zone1.Mn_i
                                        Icr_s += new_steel_zone.Icr_i
                                        top2_y = 1/3.0*c
                                        bot2_y = bot_y
                                        top2_stress_y = (c-top2_y)/c*0.003/0.002*60
                                        bot2_stress_y = (c-bot2_y)/c*0.003/0.002*60
                                        new_steel_zone2 = Steel_zone(c=c,direction = 'y',
                                        ^^Inum_ele = list[i], top = top2_y,bot = bot2_y,
                                        ^^Itop_stress = top2_stress_y, bot_stress = bot2_stress_y)
                                        Pn_s += new_steel_zone2.Pn_i
                                        Mn_s += new_steel_zone2.Mn_i
                                        Icr_s += new_steel_zone.Icr_i

                Pn = round(0.9*(Pn_s+ Pn_c))
                Mn = round(0.9*(Mn_s + Mn_c)/12)
                Icr =  (Icr_s+ Icr_c)/(12**4)

                list_Mn.append(Mn)
                list_Pn.append(Pn)
                list_Icr.append(Icr)
                list_c.append(c)
                list_all.append(c)
                list_all.append(Pn)
                list_all.append(Mn)
                list_all.append(Icr)

            for i in range(len(list_c)):
                worksheet_c_3by1.write(row,i+1,list_c[i])
                worksheet_Pn_3by1.write(row,i+1,list_Pn[i])
                worksheet_Mn_3by1.write(row,i+1,list_Mn[i])
                worksheet_Icr_3by1.write(row,i+1,list_Icr[i])
            dict_all[str(count_num)]=[list_all]
workbook_3by1.close()

######just a line (10by0)
###algorithm parameters
n_grid_x = 10
n_grid_y = 0 #grid size: n_grid_x by n_grid_y
num_zone = n_grid_y*2+1

h = n_grid_y*len_single_wall + thick_wall #372in
d = h-protection

coord_zone = []
for i in range(num_zone):
    coord_zone.append(i*len_single_wall/2.0+protection)


# Create an new Excel file and add four worksheet.
```

```
workbook_3by0 = xlsxwriter.Workbook('10by0(1)_4sheets_2.xlsx')
worksheet_c_3by0 = workbook_3by0.add_worksheet('c')
worksheet_Pn_3by0 = workbook_3by0.add_worksheet('Pn')
worksheet_Mn_3by0 = workbook_3by0.add_worksheet('Mn')
worksheet_Icr_3by0 = workbook_3by0.add_worksheet('Icr')


worksheet_c_3by0.write('A1', 'Key')
worksheet_Pn_3by0.write('A1', 'Key')
worksheet_Mn_3by0.write('A1', 'Key')
worksheet_Icr_3by0.write('A1', 'Key')
num_possible_c = 12

for i in range(num_possible_c):
    worksheet_c_3by0.write(0,i+1,'c (in)')
    worksheet_Pn_3by0.write(0,i+1,'Pn (kips)')
    worksheet_Mn_3by0.write(0,i+1,'Mn (kip.ft)')
    worksheet_Icr_3by0.write(0,i+1,'Icr (ft^4)')

list_count_num = []
row = 0
for i_1 in range(n_grid_x):
    num_1 = i_1+1
    count_num = [num_1]

    row += 1
    list_count_num.append(count_num)
    worksheet_c_3by0.write(row,0, str(count_num))
    worksheet_Pn_3by0.write(row,0, str(count_num))
    worksheet_Mn_3by0.write(row,0, str(count_num))
    worksheet_Icr_3by0.write(row,0, str(count_num))

    list = count_num

    list_Pn = []
    list_c = []
    list_Mn = []
    list_Icr =[]
    list_all = []

    for j in range(num_possible_c):
        c = j*1+0.5
        a = 0.8*c

        #concrete
        new_concrete_zone = Concrete_zone(direction = 'x', num_ele = count_num[0]/2.0,
        ^^Icoord = coord_zone[0])
        Pn_c = 0.85*concrete_strength*new_concrete_zone.area*(a-new_concrete_zone.top)
        /new_concrete_zone.height
        Mn_c = 0.85*concrete_strength*new_concrete_zone.area*(a-new_concrete_zone.top)
        /new_concrete_zone.height*(h/2.0-(a+new_concrete_zone.top)/2.0)
        Icr_c = (new_concrete_zone.area/new_concrete_zone.height)*((a-new_concrete_zone.top)**3)
        /12.0+new_concrete_zone.area/new_concrete_zone.height*(a-new_concrete_zone.top)
        *(((a+new_concrete_zone.top)/2.0-c)**2.0)

        #steel
        Pn_s = 0
        Mn_s = 0
        Icr_s = 0

        location = [2.5,h-2.5]
        if c <=3*protection: ###Top (compression) not yield, Bottom (tension) yield
            list_zone_strain = [c,2/3.0*c,d-5/3.0*c]

            for i in range(2):
                location_x = location[i]
                if location_x <= (c*5/3.0):
                    stress_x = (c-location_x)/c*0.003/0.002*60
                if location_x >(c*5/3.0):
                    stress_x = -60
                new_steel_zone = Steel_zone(c =c, direction = 'x', num_ele = list[0]/2.0,
            ^^Ilocation = location_x,stress = stress_x)
            Pn_s += new_steel_zone.Pn_i
            Mn_s += new_steel_zone.Mn_i
            Icr_s += (new_steel_zone.num_ele*new_steel_zone.length*thick_steel)
            *(new_steel_zone.location-c)**2


        if c > 3*protection and c <= 3/5.0*d:
        ###Top (compression) yield, Bottom (tension) yield
            list_zone_strain = [1/3*c,2/3*c,2/3*c,d-5/3*c]
            for i in range(2):
                location_x = location[i]
                if location_x <= (1/3.0*c):
                    stress_x = 60
                if location_x > (1/3.0*c) and location_x <= (c*5/3.0):
                    stress_x = (c-location_x)/c*0.003/0.002*60
                if location_x >(c*5/3.0):
                    stress_x = -60
                new_steel_zone = Steel_zone(c=c,direction = 'x', num_ele = list[0]/2.0,
                ^^Ilocation = location_x,stress = stress_x)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += (new_steel_zone.num_ele*new_steel_zone.length*thick_steel)
                *(new_steel_zone.location-c)**2

        if c > (3.0/5*d):

            #break

            list_zone_strain = [1/3*c,2/3.0*c,d-c]
            for i in range(2):
                location_x = location[i]
                if location_x <= (c*1/3.0):
                    stress_x = 60
                if location_x > (c*1/3.0):
                    stress_x = (c-location_x)/c*0.003/0.002*60
                new_steel_zone = Steel_zone(c=c,direction = 'x', num_ele = list[0]/2.0,
                ^^Ilocation = location_x,stress = stress_x)
                Pn_s += new_steel_zone.Pn_i
                Mn_s += new_steel_zone.Mn_i
                Icr_s += (new_steel_zone.num_ele*new_steel_zone.length*thick_steel)
                *(new_steel_zone.location-c)**2


        Pn = round(0.9*(Pn_s+ Pn_c))
        Mn = round(0.9*(Mn_s + Mn_c)/12)
        Icr =  (Icr_s+ Icr_c)/(12**4)

        list_Mn.append(Mn)
        list_Pn.append(Pn)
        list_Icr.append(Icr)
        list_c.append(c)
        list_all.append(c)
        list_all.append(Pn)
        list_all.append(Mn)
        list_all.append(Icr)

    for i in range(len(list_c)):
        worksheet_c_3by0.write(row,i+1,list_c[i])
        worksheet_Pn_3by0.write(row,i+1,list_Pn[i])
        worksheet_Mn_3by0.write(row,i+1,list_Mn[i])
        worksheet_Icr_3by0.write(row,i+1,list_Icr[i])
    dict_all[str(count_num)]=[list_all]

workbook_3by0.close()
pickle.dump( dict_all, open( "layout_data_8at10by2.p", "wb" ) )
```

A.2 Script for the fixed location extension case

```
###### This script is writen by Yu Zhang at MIT in July 2016,
###### for the fixed location extension case.

import math
import pylab as plt
import random
import numpy
import xlsxwriter
from pyDOE import *
import pickle

#####PARAMETERS

N = 5 #number of runs

### Algorithm parameters

n_pop = 2000 #population size
n_gen = 5 #generation size
rate_mut = 0.5 #mutation rate
rate_parents = 0.1 #parent individuals selected in the population
n_winner = 6 #number of winners displayed graphically
thirty_percent = int(round(n_pop*0.3))
rate_max_intersect = 0.8
rate_restricted = 0.5
rate_initial_optimal = 0.6
n_parents = int(n_pop*rate_parents)
n_mut = int(rate_mut*n_pop)

#Property of Shearwall
w_ele = 12.0/12 #width of a wall (ft)
l_net_ele = 10.0 #length of a net wall  (ft)
l_ele = l_net_ele
h_ele = 12.0 #height of a wall(ft)
area_ele = w_ele*l_ele
E = 3600*144 # Modulus of Elasticity (ksf)
G = 3000*144 # Shear Modulus (ksf)
element_A = w_ele*l_ele

#Property of Structure
n_grid_x = 12
n_grid_y = 10 #grid size: n_grid_x by n_grid_y
n_floor = 15
margin_x = 0#2*l_ele
margin_y = 0#2*l_ele
x_load = l_net_ele*n_grid_x/2.0 + margin_x  #center of load/mass
y_load = l_net_ele*n_grid_y/2.0 + margin_y
n_line_x = n_grid_x*(n_grid_y+1)
n_line_y = n_grid_y*(n_grid_x+1)
n_var = n_line_x+n_line_y
n_dots = (n_grid_x+1)*(n_grid_y+1)
dimension_x = n_grid_x*l_ele+2*margin_x  #ft
dimension_y = n_grid_y*l_ele+2*margin_y   #ft
aspect_ratio = n_floor*h_ele/max(dimension_x,dimension_y)

initial_optimal = round(rate_initial_optimal*n_var)
list_irregular_contour = [[[51,91],[59,100]],[[61,0],[120,19]]]

list_wall_free_area = [] #Ajustable based on user's needs
list_atrium = [] #Ajustable based on user's needs

#Property of RC concrete design
thick_wall = w_ele*12 #(in)12in
protection = thick_wall/2.0 #6in
len_single_wall = 10*12 #in
area_single_bar = 0.79 #in^2 #6
spacing = 10#in
layer = 2
```

```
thick_steel = (1.0*area_single_bar)/spacing*layer #in
h = n_grid_y*len_single_wall + thick_wall #372in
d = h-protection

#Load
#1.2D + 1.6W + L+ 0.5S
#Dead Load
dl_fac_ul = 1.2
dl_fac_service = 1.0

P_dl_stress = 175#psf
P_dl = dl_fac_ul*P_dl_stress*(n_floor-1)*(120*100-60*20-1*100)/1000 #kips
P_dl_service = P_dl/dl_fac_ul*dl_fac_service

#Live Load
ll_fac_ul = 1.0
ll_fac_service = 1.0
wl_fac_ul = 1.6
wl_fac_service = 1.0

P_ll_stress = 40 #psf
P_ll = ll_fac_ul*P_ll_stress*(n_floor-1)*(120*100-60*20-1*100)/1000 #kips
P_ll_service = P_ll/ll_fac_ul*ll_fac_service

P_total = (P_dl+P_ll)/2
P_total_service = (P_dl_service+P_ll_service)/2

wl_stress = 30 #psf
Q_x = wl_fac_ul*wl_stress*120*n_floor*h_ele/1000 #kips
Q_y = wl_fac_ul*wl_stress*200*n_floor*h_ele/1000 #Maximum base shear in y direction (kips)
Q_x_service = Q_x/wl_fac_ul*wl_fac_service
Q_y_service = Q_y/wl_fac_ul*wl_fac_service
Mu_x = Q_x*n_floor*h_ele/2
Mu_y = Q_y*n_floor*h_ele/2
Mu_x_service = Q_x_service*n_floor*h_ele/2
Mu_y_service = Q_y_service*n_floor*h_ele/2

n_x_wall_shear = Q_x/(10.9*len_single_wall/12+19.2*2*len_single_wall/12)
n_y_wall_shear = Q_y/(10.9*len_single_wall/12+19.2*2*len_single_wall/12)

list_fail_type = ['Larger than 3 columns','Larger than 7 rows',
   ^^I^^I^^I^^I 'Service Pu Fail','Service Mu Fail','UL Pu Fail','UL Mu Fail']

##data loading

dict_cracked_data = pickle.load( open( "layout_data_8at10by2.p", "rb" ) )

##dictionary to store the elements' coordinates:
dict_element_coord_all = {}
for i in range(n_var):
    index = i
    #get the coordinates if line is in y direction
    if index in range(n_line_y): #index from 0 to n_line_y is in y direction
        xi = l_net_ele*(index/n_grid_y)+margin_x  # the x coordinate of the starting point i
        yi = l_net_ele*(index%n_grid_y)+margin_y  # the y coordinate of the first point i
        xj = xi # the x coordinate of the ending point j
        yj = l_net_ele*((index%n_grid_y)+1)+margin_y # the y coordinate of the ending point j

    #get the coordinates if line is in x direction
    if index in range(n_line_y,n_line_x+n_line_y):
        xi = l_net_ele*((index-n_line_y)%n_grid_x)+margin_x
        yi = l_net_ele*((index-n_line_y)/n_grid_x)+margin_y
        xj = l_net_ele*(((index-n_line_y)%n_grid_x)+1)+margin_x
        yj = yi

    coordi = [xi,yi]
    coordj = [xj,yj]

    dict_element_coord_all[str(index)] = [coordi,coordj]

list_element_fixed = [0,2,3,5,7,8,20,21,27,29,33,37,39,43,44,56,57,58,59,60,61,62,63,
^^I^^I^^I^^I 64,66,67,68,69,82,83,84,96,97,98,99,102,103,104,122,123,127,129,
^^I^^I^^I^^I 130,131,132,135,147,159,161,162,165,166,167,168,169,170,182,186,
^^I^^I^^I^^I 189,194,198,201,202,203,204,212,213,225,226,227,228,251,254,256,
^^I^^I^^I^^I 258,259,261,243] #These are picked according to the fixed floorplan

dict_element_coord = {}
for i in range(len(list_element_fixed)):
    dict_element_coord[str(i)]= dict_element_coord_all[str(list_element_fixed[i])]
n_var = len(list_element_fixed)

#####  CLASS

class VoidBlock:
    def __init__(self, list_coord = None ): ######remember to consider EI
        self.list_coord = list_coord
        if len(list_coord) == 3:
            self.shape = 'Triangle'
        if len(list_coord) == 2:
            self.shape = 'Rectangle'
            self.coord_bl = list_coord[0]
            self.coord_tr = list_coord[1]
            self.coord_br = [list_coord[1][0],list_coord[0][1]]
            self.coord_tl = [list_coord[0][0],list_coord[1][1]]
            self.dimx = self.coord_tr[0]-self.coord_bl[0]
            self.dimy = self.coord_tr[1]-self.coord_bl[1]
            self.centerx = (self.coord_tr[0]+self.coord_bl[0])/2
            self.centery = (self.coord_tr[1]+self.coord_bl[1])/2
            self.plot_x = [self.coord_bl[0],self.coord_br[0],self.coord_tr[0],self.coord_tl[0],
            ^^Iself.coord_bl[0]]
            self.plot_y = [self.coord_bl[1],self.coord_br[1],self.coord_tr[1],self.coord_tl[1],
            ^^Iself.coord_bl[1]]

            self.area = self.dimx*self.dimy
            self.list_index = []
            for element in list_elements:
                if element.xi in range(self.coord_bl[0],self.coord_tr[0]+1)
                or element.xj in range(self.coord_bl[0],self.coord_tr[0]+1):
                    if element.yi in range(self.coord_bl[1],self.coord_tr[1]+1)
                    or element.yj in range(self.coord_bl[1],self.coord_tr[1]+1):
                        self.list_index.append(element.index)

            self.num_elements = len(self.list_index)

#each line in ground structure is an element of lateral system
class Element:
    def __init__(self, index):
        self.index = index  ##index is something like 3,6,19,37...
        self.xi = dict_element_coord[str(index)][0][0]  # the x coordinate of the starting point i
        self.yi = dict_element_coord[str(index)][0][1]  # the y coordinate of the first point i
        self.xj = dict_element_coord[str(index)][1][0] # the x coordinate of the ending point j
        self.yj = dict_element_coord[str(index)][1][1] # the y coordinate of the ending point j
        #get the coordinates if line is in y direction
        if self.xi == self.xj: #index from 0 to n_line_y is in y direction
            self.direction = 'y'
            self.dimx = w_ele #dimension in x direction
            self.dimy = l_net_ele

        #get the coordinates if line is in x direction
        if self.yi == self.yj:
            self.direction = 'x'
            self.dimx = l_net_ele
            self.dimy = w_ele

        self.ix = (self.dimx*(self.dimy**3))/12 #inertia about the x axis
        self.iy = (self.dimy*(self.dimx**3))/12 #inertia about the y axis

        #make its coordinates to a property
        self.coordi = [self.xi,self.yi]
        self.coordj = [self.xj,self.yj]
        self.centerx = (self.xi + self.xj)/2.0
        self.centery = (self.yi + self.yj)/2.0
    def classify_quadrant(self, x_mass, y_mass):
        if self.centerx > x_mass and self.centery >= y_mass:
            self.quadrant = '1'
            list_ele_quadrant_1.append(index)
        elif self.centerx <= x_mass and self.centery > y_mass:
            self.quadrant = '2'
            list_ele_quadrant_2.append(index)
        elif self.centerx < x_mass and self.centery <= y_mass:
            self.quadrant = '3'
            list_ele_quadrant_3.append(index)
        elif self.centerx >= x_mass and self.centery < y_mass:
            self.quadrant = '4'
            list_ele_quadrant_4.append(index)
        elif self.centerx == x_mass and self.centery == y_mass:
            self.quadrant = 'center'
            list_ele_quadrant_cen.append(index)


list_elements = []
list_ele_quadrant_1 = []
list_ele_quadrant_2 = []
list_ele_quadrant_3 = []
list_ele_quadrant_4 = []
list_ele_quadrant_cen = []
for i in range(n_var):
    element = Element(i)
    list_elements.append(element)
list_void_block_classes = []
list_void_index = []
area_times_coord_x = (dimension_x*dimension_y)*x_load
area_times_coord_y = (dimension_x*dimension_y)*y_load
area = dimension_x*dimension_y

list_void_block = list_irregular_contour +list_wall_free_area + list_atrium

for block in list_void_block:
    new_void_block = VoidBlock(block)
    list_void_block_classes.append(new_void_block)
    if block not in list_wall_free_area:
        area_times_coord_x -= new_void_block.area*new_void_block.centerx
        area_times_coord_y -= new_void_block.area*new_void_block.centery
        area -= new_void_block.area

x_mass = 1.0*area_times_coord_x/area
y_mass = 1.0*area_times_coord_y/area


class Part:
    def __init__(self, index_list):
        self.index_list = index_list #the indexes in this part
        #self.cmx = 0
        #self.cmy = 0
        self.Iy = 0
        self.Ix = 0
        self.stress_x = 0
        self.stress_y = 0
        self.Icr = []
        self.A = len(index_list)*area_ele

    def find_center_of_mass(self,dict_el):
        centerx_all = 0
        centery_all = 0
        for i in range(len(self.index_list)):
            element = dict_el[str(self.index_list[i])]
            centerx_all += element.centerx
            centery_all += element.centery

        self.cmx = centerx_all/len(self.index_list)
        self.cmy = centery_all/len(self.index_list)
```

```
def calculate_I(self, dict_el):
    self.find_center_of_mass(dict_el)
    for index in self.index_list:
        element = dict_el[str(index)]
        self.Ix += element.ix + w_ele*l_ele*((element.centery - self.cmy)**2)
        self.Iy += element.iy + w_ele*l_ele*((element.centerx - self.cmx)**2)

def count_num(self, dict_el):
    if len(self.index_list) == 1:
        element = dict_el[str(self.index_list[0])]
        self.dimx = element.dimx
        self.dimy = element.dimy
        self.centerx = element.centerx
        self.centery = element.centery
        if self.dimx >self.dimy:
            self.num_zone_y = 1
            self.count_num_y = [1]
            self.num_zone_x = 3
            self.count_num_x = [0,1,0]
        else:
            self.num_zone_x = 1
            self.count_num_x = [1]
            self.num_zone_y = 3
            self.count_num_y = [0,1,0]

    else:
        list_x = []
        list_y = []
        list_centerx = []
        list_centery = []
        for index in self.index_list:
            element = dict_el[str(index)]
            list_x.append(element.xi)
            list_x.append(element.xj)
            list_y.append(element.yi)
            list_y.append(element.yj)
            list_centerx.append(element.centerx)
            list_centery.append(element.centery)
        self.dimx = max(list_x)- min(list_x)
        self.dimy = max(list_y)- min(list_y)
        self.num_zone_x = int((self.dimx/l_ele)*2+1)
        self.num_zone_y = int((self.dimy/l_ele)*2+1)
        self.count_num_x = []
        self.count_num_y = []
        for i in range(self.num_zone_x):
            count_num_i = 0
            for centerx in list_centerx:
                if centerx == min(list_x)+i*l_ele/2:
                    count_num_i +=1
            self.count_num_x.append(count_num_i)
        for i in range(self.num_zone_y):
            count_num_i = 0
            for centery in list_centery:
                if centery == min(list_y)+i*l_ele/2:
                    count_num_i +=1
            self.count_num_y.append(count_num_i)

def strength_and_Icr(self,count_num,Pu,Mu,Pu_service, Mu_service):
    if len(count_num) <= 7:
        for j in range(len(count_num)):
            if  count_num[j]>(j%2+3):
                strength_check = 'Larger than 3 columns'
                break
            else:
                strength_check = 'Continue'
        if strength_check == 'Continue':
            ###in x/y direction:
            cracked_data = dict_cracked_data[str(count_num)][0]
            for i in range(len(cracked_data)/4-1):
                if min(cracked_data[i*4+1],cracked_data[i*4+5])<= Pu
                and max(cracked_data[i*4+1],cracked_data[i*4+5]) > Pu:
                    if Mu <= (cracked_data[i*4+2]+cracked_data[i*4+6])/2:
                        strength_check = 'Continue'
                        break
                    else:
                        strength_check = 'UL Mu Fail'
                else:
                    strength_check = 'UL Pu Fail'

            if strength_check == 'Continue':
                for i in range(len(cracked_data)/4-1):
                    if min(cracked_data[i*4+1],cracked_data[i*4+5])<= Pu_service
                    and max(cracked_data[i*4+1],cracked_data[i*4+5]) > Pu_service:
                        if Mu_service <= (cracked_data[i*4+2]+cracked_data[i*4+6])/2:
                            strength_check = 'Pass'
                            self.Icr.append((cracked_data[i*4+3]+cracked_data[i*4+7])/2.0)
                            #self.c = (cracked_data_x[i*4]+cracked_data_x[i*4+4])/2
                            break
                        else:
                            strength_check = 'Service Mu Fail'

                    else:
                        strength_check = 'Service Pu Fail'

        else:
            strength_check = 'Larger than 7 rows'

        return strength_check

class Layout:
    def __init__(self, string): #g_pop_i is a binary string which representing one layout
        self.list_index_el = [] #list of indeces for the elements that are active
        self.list_el = [] #list of elements(these are classes) that are active
        self.dict_el ={}
        self.list_parts =[]
        self.list_coord = []
        self.list_coord_index = []
        self.list_parts_all = []
        self.list_overlap_coord_set = []
        self.string = string
        self.adjust = 'Major'
        self.material_factor = 1## was "material_factor"

        for j in range(len(self.string)): #self.string is a binary string which representing one layout
            if j in [13,14,16,17,25,26,55,56,59]:
                self.string = self.string[:j]+'1'+self.string[j+1:]
            if self.string[j] =='1': #to see if the element at the position is avtive
            #if active, add its position index to the active list of this layout
                self.list_index_el.append(j)
                new_element = Element(j) #generate a new element with the index j
                new_element.classify_quadrant(x_mass,y_mass)
                #add the element to the list of active elements of this layout
                self.list_el.append(new_element)
                self.dict_el[str(j)] = new_element #also a dictionary

        self.len_list_el = len(self.list_el)

        self.form_connected_parts()
        self.calculate_center_of_stiffness()

        self.generate_material_slop() # get the structural weight and its slop
        self.p1 = self.penalty_on_torsion()  # generate p1
        #penalty on distribution,disabled for the fixed floorplan case
        #self.p6 = penalty_dis(self.list_el)

        self.p6 = 0
        self.check_strength_and_drift()
        if self.strength_check_total == 'Fail':
            self.p4 = 1 #was 100000
            self.p5 = 0.5 #was -1
        if self.strength_check_total == 'Pass':
            self.p4 = 0
            if self.drift_check == 'Pass':
                self.p5 = 0
            if self.drift_check == 'Fail':
                self.p5 = 1 #was 50000
        self.p2 = 0  # shear simplified

        self.p3 = self.penalty_overlap() #penalty on overlap coords
        if   self.p2 + self.p3 +self.p6 > 1:
            self.constraint = 10000#constraint
        else:
            self.constraint = 0

        if self.p4 == 1:
            self.constraint += 10000
            self.p4 = 10

        self.fitness = (self.weight+self.constraint)*(( 5*self.material_factor+self.p1
        ^^I^^I^^I^^I+self.p2+ self.p3 +self.p4 +self.p5 +1*self.p6 +self.p7)**2)

    def generate_material_slop(self):
        #preferred_num = rate_min_ele*n_var
        #max_num = rate_max_ele*n_var
        n_active = len(self.list_index_el)
        self.weight = n_active

        if self.weight < 0.3*n_var:
            self.material_factor = 1

        ###test
        if 1.0*n_active > initial_optimal:
            self.material_factor = 1+ 50*(1.0*n_active/n_var - rate_initial_optimal)
    def penalty_on_torsion(self):
        self.distance_cs_cm = math.sqrt((self.center_stiff_x-x_mass)**2
        ^^I^^I^^I^^I^^I^^I^^I+(self.center_stiff_y-y_mass)**2) #ft
        penalty = 1.0*self.distance_cs_cm/preferred_distance
        return penalty

    def check_shear_strength(self):
        n_y_shear = sum(i < n_line_y for i in self.list_index_el)
        n_x_shear = sum(i >= n_line_y for i in self.list_index_el)
        if n_x_shear <= n_x_wall_shear or n_y_shear <= n_y_wall_shear:
            self.shear_strength_check = 'Fail'
        else:
            self.shear_strength_check = 'Pass'

    #method that can plot the layout
    def plotlayout(self, subfigure, fitness = None, figure = 1, cs0 = [0,0], cl = [x_mass,y_mass] ):
        fig =plt.figure(figure, figsize = (20,20))
        plt.subplot(6, 5, subfigure)
        plt.scatter(cs0[0],cs0[1], c = "k",marker ='*',  s=100)
        plt.scatter(cl[0],cl[1], c = "k",marker = 'x',  s=100)

        for element in list_elements:
            x = [element.xi,element.xj]
            y = [element.yi,element.yj]
            fig =plt.figure(figure, figsize = (20,20))
            plt.plot(x,y,'#D3D3D3',linewidth = 2)
        for element in self.list_el:
            x = [element.xi,element.xj]
            y = [element.yi,element.yj]
            fig =plt.figure(figure, figsize = (20,20))

            plt.plot(x,y,'k',linewidth=2)

        for block in list_void_block_classes:
            if block.list_coord in list_atrium:
                plt.subplot(6, 5, subfigure)
                plt.plot(block.plot_x,block.plot_y,'k')
                plt.plot([block.list_coord[0][0],block.list_coord[1][0]],
                ^^I^^I [block.list_coord[0][1],block.list_coord[1][1]],'k')
                plt.plot([block.list_coord[0][0],block.list_coord[1][0]],
            ^^I^^I [block.list_coord[1][1],block.list_coord[0][1]],'k')

    plt.plot([0,0,50,50,60,60,120,120,60,60,0],[0,100,100,90,90,100,100,20,20,0,0],
    ^^I'#D3D3D3',linewidth=1)
    plt.xlim(-10,dimension_x+10)
    plt.ylim(-10,dimension_y+10)

    plt.title("Fit:"+str(round(fitness*10)/10.0)+', N:'+str(self.weight)+', S:'+str(self.check))
    frame1 = plt.gca()

    frame1.axes.get_xaxis().set_visible(False)
    frame1.axes.get_yaxis().set_visible(False)
    frame1.set_frame_on(False)

    plt.savefig('figure'+str(figure)+'.svg')

def get_total_IandA (self):
    self.A_total = 0
    self.A_x = 0
    self.A_y = 0
    self.I_total_x = 0
    self.I_total_y = 0

    for part in self.list_parts:

        self.A_total += part.A
        self.I_total_x += part.Ix
        self.I_total_y += part.Iy

    for element in self.list_el:
        if element.direction == 'x':
            self.A_x += element_A
        if element.direction == 'y':
            self.A_y += element_A

def form_list_coordinates(self):
    for element in self.list_el:
        self.list_coord_index.append([element.coordi,element.index])
        self.list_coord_index.append([element.coordj,element.index])
        self.list_coord.append(str(element.coordi))
        self.list_coord.append(str(element.coordj))

def form_overlap_coord_set(self):
    self.list_coord_set = list(set(self.list_coord))
    self.list_overlap_coord_set = []

    for coord in self.list_coord_set:
        if self.list_coord.count(coord)>1:
            self.list_overlap_coord_set.append(coord) ##could be used to find conncetion

def form_overlap_coord(self):
    self.list_coord_set = list(set(self.list_coord))
    self.list_overlap_coord = []

    for coord in self.list_coord_set:
        if self.list_coord.count(coord)>1:
            for i in range(self.list_coord.count(coord)):
                self.list_overlap_coord.append(coord) #find the overlapped times of a coord

####penalty 3 is for overlap

def penalty_overlap(self):
    num_overlap = 0
    for part in self.list_parts:
        list_part_transformed_coord = []
        for index in part.index_list:
            transformed_num_1 = dict_element_coord[str(index)][0][0]
            ^^I*1000+dict_element_coord[str(index)][0][1]
            transformed_num_2 = dict_element_coord[str(index)][1][0]
            ^^I*1000+dict_element_coord[str(index)][1][1]
            list_part_transformed_coord.append(transformed_num_1)
```

```
                list_part_transformed_coord.append(transformed_num_2)
            differece = len(part.index_list) - len(list(set(list_part_transformed_coord)))
            if differece < 0:
                penalty = 0
            if differece >=0:
                penalty = 1  #
        return penalty

    def form_connected_parts(self):
        self.form_list_coordinates()
        self.form_overlap_coord_set()
        list_connected = []
        list_connected_all = []
        list_not_connected =[]
        for j in range(len(self.list_overlap_coord_set)):
            list_one_coord_connect = []
            for element in self.list_el:
                if str(element.coordi) == self.list_overlap_coord_set[j]
                or str(element.coordj)  == self.list_overlap_coord_set[j]:
                ^^I#record only the index instead of coordinates
                    list_connected_all.append(element.index)
                    list_one_coord_connect.append(element.index)
            list_connected.append(list_one_coord_connect)

        list_connected_all_set = list(set(list_connected_all))
        for j in self.list_index_el:
            if j not in list_connected_all_set:
                list_j = [j]
                list_not_connected.append(list_j)

        while len(list_connected_all) > len(list_connected_all_set):
            for item in list_connected_all_set:
                if list_connected_all.count(item)>1: #this element index has showed up twice
                ^^I^^I^^I^^I^^I^^I^^I^^I #-> should be connected
                    list_connecting_parts = []
                    list_index_delete = []

                    list_connected_1 = list_connected
                    list_connected_all_1 = list_connected_all

                    for j in range(len(list_connected)):

                        if item in list_connected[j]:

                            list_index_delete.append(j)
                            for k in range(len(list_connected[j])):

                                list_connecting_parts.append(list_connected[j][k])

                    list_index_delete.sort(reverse = True)
                    for index in list_index_delete:
                        list_connected.remove(list_connected[index])
                    if list_connected_all.count(item)>1:
                        list_connected_all.remove(item)

                    list_connecting_parts_1 = list(set(list_connecting_parts))
                    if list_connecting_parts_1 != []:
                        list_connected.append(list_connecting_parts_1)

                    else:
                        print 'overlap set_2',len(list_connected_all)
                        print 'set', len(list_connected_all_set)
                        print item
                        print 'C_1',list_connected_1
                        print 'C_all',list_connected_all_1


        self.list_parts_all = list_connected + list_not_connected
        self.list_parts =[]

        for part_index in self.list_parts_all:
            new_part = Part(part_index)
            new_part.calculate_I(self.dict_el)
            new_part.count_num(self.dict_el)
            self.list_parts.append(new_part)

    def calculate_center_of_stiffness(self):
        sum_Ix_y = 0
        sum_Iy_x = 0
        sum_Iy = 0
        sum_Ix = 0
        for part in self.list_parts:
            sum_Ix_y += part.Ix*part.cmy
            sum_Iy_x += part.Iy*part.cmx
            sum_Iy += part.Iy
            sum_Ix += part.Ix
        if sum_Iy == 0 or sum_Ix == 0:
            self.center_stiff_x = 0
            print "I = 0"
            self.center_stiff_y = 0
        else:
            self.center_stiff_x = sum_Iy_x/sum_Iy
            self.center_stiff_y = sum_Ix_y/sum_Ix

        ### Now we add the property of quadrant
        if self.center_stiff_x > x_mass and self.center_stiff_y >= y_mass:
            self.quadrant = '1'
        elif self.center_stiff_x <= x_mass and self.center_stiff_y > y_mass:
            self.quadrant = '2'
        elif self.center_stiff_x < x_mass and self.center_stiff_y <= y_mass:
            self.quadrant = '3'
        elif self.center_stiff_x >= x_mass and self.center_stiff_y < y_mass:
            self.quadrant = '4'
        elif self.center_stiff_x == x_mass and self.center_stiff_y == y_mass:
            self.quadrant = 'center'

        ###classify element in quadrant

        self.list_ele_quadrant_1 = []
        self.list_ele_quadrant_2 = []
        self.list_ele_quadrant_3 = []
        self.list_ele_quadrant_4 = []
        self.list_ele_quadrant_cen = []

        for element in self.list_el:
            if element.quadrant == '1':
                self.list_ele_quadrant_1.append(element.index)
            elif element.quadrant == '2':
                self.list_ele_quadrant_2.append(element.index)
            elif element.quadrant == '3':
                self.list_ele_quadrant_3.append(element.index)
            elif element.quadrant == '4':
                self.list_ele_quadrant_4.append(element.index)
            elif element.quadrant == 'center':
                self.list_ele_quadrant_cen.append(element.index)


    def check_strength_and_drift(self):
        self.get_total_IandA()
        if self.I_total_x == 0:
            print len(self.list_parts)
        list_strength_check =[]
        for part in self.list_parts:

            mu_x_i = Mu_x*part.Iy/self.I_total_y
            mu_y_i = Mu_y*part.Ix/self.I_total_x
            mu_x_service_i = Mu_x_service*part.Iy/self.I_total_y
            mu_y_service_i = Mu_y_service*part.Ix/self.I_total_x
            pu_i = P_total*part.A/self.A_total
            pu_service_i = P_total_service*part.A/self.A_total

            count_num_x = part.count_num_x
        count_num_y = part.count_num_y
        strength_check_1 = part.strength_and_Icr(
        ^^I^^I^^I^^I   count_num_x,pu_i,mu_x_i,pu_service_i, mu_x_service_i)
        strength_check_2 = part.strength_and_Icr(
        ^^I             count_num_y,pu_i,mu_y_i,pu_service_i, mu_y_service_i)
        strength_check_3 = part.strength_and_Icr(
        ^^I^^I^^I^^I   list(reversed(count_num_x)),pu_i,mu_x_i,pu_service_i, mu_x_service_i)
        strength_check_4 = part.strength_and_Icr(
        ^^I             list(reversed(count_num_x)),pu_i,mu_y_i,pu_service_i, mu_y_service_i)
        list_strength_check.append(strength_check_1)
        list_strength_check.append(strength_check_2)
        list_strength_check.append(strength_check_3)
        list_strength_check.append(strength_check_4)
    num_fail = len(list_strength_check) - list_strength_check.count('Pass')


    self.num_UL_Pu = list_strength_check.count('UL Pu Fail')
    self.num_UL_Mu = list_strength_check.count('UL Mu Fail')
    self.num_SVS_Pu = list_strength_check.count('Service Pu Fail')
    self.num_SVS_Mu = list_strength_check.count('Service Mu Fail')
    self.strength_fail = num_fail

    #####
    self.dimension_fail = list_strength_check.count('Larger than 3 columns')
     ^^I^^I^^I^^I  +list_strength_check.count('Larger than 7 rows')
    if self.dimension_fail > 0:
        self.p7 = 1
    else:
        self.p7 = 0
    #####

    if num_fail == 0:
        self.strength_check_total = 'Pass'
    else:
        self.strength_check_total = 'Fail'

    if self.strength_check_total == 'Pass':
        Icr_1 = 0
        Icr_2 = 0
        Icr_3 = 0
        Icr_4 = 0
        for part in self.list_parts:
            Icr_1 += part.Icr[0]
            Icr_2 += part.Icr[1]
            Icr_3 += part.Icr[2]
            Icr_4 += part.Icr[3]
        self.Icr_x = min(Icr_1,Icr_3)
        self.Icr_y = min(Icr_2,Icr_4)
        u_b_x = drift_bending(self.Icr_x,Q_x_service)
        u_b_y = drift_bending(self.Icr_y,Q_y_service)
        u_s_x = drift_shear(self.A_x,Q_x_service)
        u_s_y = drift_shear(self.A_y,Q_y_service)
        self.u_x = u_b_x+u_s_x
        self.u_y = u_b_y+u_s_y
        self.drift_x = round(h_ele/self.u_x)
        self.drift_y = round(h_ele/self.u_y)
        if self.drift_x <= 400 or self.drift_y <=400:
            self.drift_check = 'Fail'
            self.check = 'Fail'
        else:
            self.drift_check = 'Pass'
            self.check = 'Pass'
    else:
        self.check = 'Fail'
        self.drift_check = 'N/A'
        self.Icr_x = 0
        self.Icr_y = 0
        self.drift_x = 'N/A'
        self.drift_y = 'N/A'
    self.ratio_I_x = round(100*(1.0*self.Icr_x/self.I_total_x))/100.0
    self.ratio_I_y = round(100*(1.0*self.Icr_y/self.I_total_y))/100.0
```

```
##### FUNCTIONS

#function that makes the number binary
# and get rid of the '0b' at the beginning
def binary(x):
    x = bin(x)
    x = x[2:]
    x = x.zfill(n_var)
    return x

### Drift
#Drift due to bending  U(bh) = Qh^3/3EI
def drift_bending (I,Q):
    if I == 0:
        u_b = 1000
    else:
        u_b = Q*(n_floor*h_ele**3)/(3*E*I) #(ft)
    return u_b

#Drift due to shear
def drift_shear (A,Q):
    if A == 0:
        u_s = 1000
    else:
        u_s = Q*(n_floor*h_ele)/(G*(5/6.0)*A) #(ft) A -> 5/6A
    return u_s

#####Penalty
#penalty1 is for the total number of elements (material limit)
#penalty2 is for the distribution of elements
def penalty_dis(list_el):
    penalty = 0
    for element in list_el:

        if abs(element.centerx-x_mass) <= acceptable_distribution_x:
            penalty += 0
        else:
            penalty += (abs(element.centerx-x_mass) - acceptable_distribution_x)
                       /(dimension_x/2 - acceptable_distribution_x)
        if abs(element.centery-y_mass) <= acceptable_distribution_y:
            penalty += 0
        else:
            penalty += (abs(element.centery-y_mass) - acceptable_distribution_y)
            ^^I^^I^^I/(dimension_y/2 - acceptable_distribution_y)
    if len(list_el)== 0:
        penalty = 100000
    else:
        penalty = 3*penalty/(len(list_el))
    return penalty

def diversity_filter(list_candidate,number_selected):
    list_selected = []
    list_selected.append(list_candidate[0])
    list_candidate = list_candidate[1:]
    for i in list_candidate:
        if len(list_selected) >= number_selected:
            break
        count = 0
        j = len(list_selected)
        for jj in range(j):
            if (len(set(list_selected[jj].list_index_el).intersection(
            ^^Iset(i.list_index_el)))*1.0/len(i.list_index_el) < rate_max_intersect)
            ^^Iand (len(list_selected) < number_selected):
                count += 1
        if count == j:
            list_selected.append(i)

    return list_selected
```

```
#select the parents from top layouts
def form_parents(list_layout):
    g_parents = diversity_filter(list_layout,n_parents)
    return g_parents

#breeding and evolve through generation

def breeding(g_parents) :
    ### classify parents by their quadrants and adjust type:
    list_parent_quadrant_1 = []
    list_parent_quadrant_2 = []
    list_parent_quadrant_3 = []
    list_parent_quadrant_4 = []
    list_parent_quadrant_cen = []
    for parent in g_parents:
        if parent.quadrant == '1':
            list_parent_quadrant_1.append(parent)
        elif parent.quadrant == '2':
            list_parent_quadrant_2.append(parent)
        elif parent.quadrant == '3':
            list_parent_quadrant_3.append(parent)
        elif parent.quadrant == '4':
            list_parent_quadrant_4.append(parent)
        elif parent.quadrant == 'center':
            list_parent_quadrant_cen.append(parent)

    lists_parent_quadrant = [list_parent_quadrant_cen,list_parent_quadrant_1,
    ^^I^^I^^I^^I^^Ilist_parent_quadrant_2,list_parent_quadrant_3,list_parent_quadrant_4]

    g_child=[]
    #breeding
    for i1 in range(int(n_pop/2)):
        children =produce_2_children(g_parents,lists_parent_quadrant)
        g_child.append(children[0])
        g_child.append(children[1])
    g_children = []
    for i1 in range(len(g_child)):
        g_children.append(g_child[i1])

    return g_children

def produce_2_children(g_parents,lists_parent_quadrant):

    first_parent = random.choice(g_parents)
    second_parent = random.choice(g_parents)

    parents = [first_parent,second_parent]

    split_point = random.randint(0,n_var-1)

    child1string= parents[0].string[:split_point]+parents[1].string[split_point:]
    child2string =parents[1].string[:split_point]+parents[0].string[split_point:]

    if '1' not in child1string:
        add_point = random.randint(0,n_var-2)
        child1string = child1string[:add_point]+'1'+child1string[add_point+1:]
    if '1' not in child2string:
        add_point = random.randint(0,n_var-2)
        child2string = child2string[:add_point]+'1'+child2string[add_point+1:]

    child1 = Layout(child1string)
    ###
    #child1.form_connected_parts()
    child2 = Layout(child2string)
    children = [child1, child2]

    return children

#mutation:
def mute(x,quadrant):
    position_mut = [random.randint(0,n_var-1) for _ in range(random.randint(1,n_var))]
    for index_mutpoint in position_mut:
        if x.string[index_mutpoint] =='1':
            x.string = x.string[:index_mutpoint]+'0'+x.string[index_mutpoint+1:]
        else:
            x.string = x.string[:index_mutpoint]+'1'+x.string[index_mutpoint+1:]
    x_update = Layout(x.string)
    return x_update

    """
    if quadrant == '1':
        list_position_deactivate = x.list_ele_quadrant_1
        list_position_activate = list(set(list_ele_quadrant_3)
        ^^I^^I^^I^^I^^I^^I .difference(set(x.list_ele_quadrant_3)))
    elif quadrant == '2':
        list_position_deactivate = x.list_ele_quadrant_2
        list_position_activate = list(set(list_ele_quadrant_4)
        ^^I^^I^^I^^I^^I^^I.difference(set(x.list_ele_quadrant_4)))
    elif quadrant == '3':
        list_position_deactivate = x.list_ele_quadrant_3
        list_position_activate = list(set(list_ele_quadrant_1)
        ^^I^^I^^I^^I^^I^^I.difference(set(x.list_ele_quadrant_1)))
    elif quadrant == '4':
        list_position_deactivate = x.list_ele_quadrant_4
        list_position_activate = list(set(list_ele_quadrant_2)
        ^^I^^I^^I^^I^^I^^I .difference(set(x.list_ele_quadrant_2)))
    elif quadrant == 'center':
        list_position_deactivate = range(1,n_var)
        list_position_activate = range(1,n_var)

    position_mut_activate = random.sample(list_position_activate,
    ^^I^^I^^I^^I^^I^^Irandom.randint(1,len(list_position_activate)))
#min(max(5,int(0.1*len(list_position_activate))),len(list_position_activate))))

    if len(list_position_deactivate) == 0:
        deactivate = 'No'
    else:
        deactivate = 'Yes'
        position_mut_deactivate = random.sample(list_position_deactivate,
        ^^I^^I^^I^^I^^I^^I  random.randint(1,len(list_position_deactivate)))
    #position_mut = [random.randint(0,n_var-1) for _ in range(random.randint(1,n_var))]
    if deactivate == 'Yes':
        for index_mutpoint in position_mut_deactivate:

            if index_mutpoint < n_var-1:
                x.string = x.string[:index_mutpoint]+'0'+x.string[index_mutpoint+1:]

    for index_mutpoint in position_mut_activate:

        if index_mutpoint < n_var-1:
            x.string = x.string[:index_mutpoint]+'1'+x.string[index_mutpoint+1:]

    if '1' not in x.string:
        add_point = random.randint(0,n_var-2)
        x.string = x.string[:add_point]+'1'+x.string[add_point+1:]

    x_update = Layout(x.string)
    return x_update
    """

    #if len(list_position_deactivate) == 0:
    #    deactivate = 'No'
    #else:
    #    deactivate = 'Yes'

    #position_mut = [random.randint(0,n_var-1) for _ in range(random.randint(1,n_var))]
    #if deactivate == 'Yes':
    """
    for index_mutpoint in position_mut_deactivate:
        if index_mutpoint < n_var-1:
            x.string = x.string[:index_mutpoint]+'0'+x.string[index_mutpoint+1:]
    for index_mutpoint in position_mut_activate:

        if index_mutpoint < n_var-1:
            x.string = x.string[:index_mutpoint]+'1'+x.string[index_mutpoint+1:]
    x = Layout(x.string)
    return x
    """

def mutation(g_children):

    index_sel_mut = [random.randint(0,len(g_children)-1) for _ in range(n_mut)]
    for index_mut in index_sel_mut:
        g_children[index_mut] = mute(g_children[index_mut],g_children[index_mut].quadrant)

    return g_children

###### Main program Loop:

def main_loop(figure_i,constraint,weight_sig,torsion_sig,distribution_sig):
    list_all_winner = []

    for generation_num in range(n_gen):
        print  generation_num

        ####Initialize the list for layouts

        if generation_num == 0:
            list_layout = [] #For the first iteration
            #randomly select n_pop individuals
            n_restricted_var = round(rate_restricted*n_var)
            for individual_index in range(10*n_pop):
                n_shear_wall = random.randint(int(0.6*n_var),n_var)
                list_index_shear_wall = random.sample(range(n_var), n_shear_wall)
                string = ''
                for digit in range(n_var):
                    if digit in list_index_shear_wall:
                        string += '1'
                    else:
                        string += '0'

                new_layout = Layout(string)
                list_layout.append(new_layout)

            #Sort the layouts by their fitness (The lower, the better, the former)
            list_layout.sort(key=lambda x: x.fitness, reverse=False)

            ####record the fitness of top10/top30%/all
            top_10_fit_list = []
            top_30_fit_list = []
            all_fit_list = []
            fit_list = []
            top_10_mat_list = []
            top_30_mat_list = []
            all_mat_list = []
            material_list = []
            top_5_mat_list = []
            top_5_mat_list_2 = []
            for layout in list_layout:
                fit_list.append(layout.fitness)
                material_list.append(layout.len_list_el)

            top_10_fit_list.append(numpy.mean(fit_list[:100]))
            top_30_fit_list.append(numpy.mean(fit_list[:10*thirty_percent]))
            all_fit_list.append(numpy.mean(fit_list))

            top_10_mat_list.append(numpy.mean(material_list[:100]))

            top_30_mat_list.append(numpy.mean(material_list[:10*thirty_percent]))
            all_mat_list.append(numpy.mean(material_list))

            list_layout = list_layout[:n_pop]
        winners = diversity_filter(list_layout,5)
        material_count =0
        material_count_2 =0
        for winner in winners:
            list_all_winner.append(winner)
            material_count += winner.len_list_el/5.0
            material_count_2 += winner.weight/5.0
        top_5_mat_list.append(material_count)
        top_5_mat_list_2.append(material_count_2)

    #choose parents
    g_parents = form_parents(list_layout)

    #breeding and  evolve through generation
    g_children = breeding(g_parents)

    #mutation:
    g_children = mutation(g_children)

    #Sort the layouts by their fitness (The lower, the better, the former)

    list_layout =  g_children +g_parents
    list_layout.sort(key=lambda x: x.fitness, reverse=False)
    list_layout = list_layout[:n_pop]
    fit_list = []
    material_list = []
    for layout in list_layout:
        fit_list.append(layout.fitness)
        material_list.append(layout.len_list_el)
    top_10_fit_list.append(numpy.mean(fit_list[:10]))
    top_30_fit_list.append(numpy.mean(fit_list[:thirty_percent]))
    all_fit_list.append(numpy.mean(fit_list))

    top_10_mat_list.append(numpy.mean(material_list[:5]))
    top_30_mat_list.append(numpy.mean(material_list[:thirty_percent]))
    all_mat_list.append(numpy.mean(material_list))

    winners = diversity_filter(list_layout,5)
    material_count =0
    material_count_2 += 0
    string = '1'*119

    for winner in winners:
        list_all_winner.append(winner)
        material_count += winner.len_list_el/5.0
        material_count_2 += winner.weight/5.0

    top_5_mat_list.append(material_count)
    top_5_mat_list_2.append(material_count_2)

###plot the 6 winners

list_col = ['O','P','Q','R','S','T']
winner = list_all_winner
for i in range(len(list_all_winner)):

    winner[i].plotlayout(i+1, fitness = winner[i].fitness, figure = figure_i,
    ^^I^^I^^I^^Ics0 = [winner[i].center_stiff_x, winner[i].center_stiff_y])
    ###newly added end
    if winner[i].p5 >= 1:
        print i ,' p5 ', winner[i].p5
    if winner[i].p4 >= 1:
        print i ,' p4 ', winner[i].num_UL_Pu

worksheet.write('B'+str(row),winner[25].string ) #sig_cs
worksheet.write('C'+str(row),winner[25].material_factor )
worksheet.write('L'+str(row),winner[25].I_total_x)
worksheet.write('M'+str(row),winner[25].Icr_x)
worksheet.write('N'+str(row),winner[25].ratio_I_x)
worksheet.write('O'+str(row),winner[25].I_total_y)
worksheet.write('P'+str(row),winner[25].Icr_y)
worksheet.write('Q'+str(row),winner[25].ratio_I_y)
worksheet.write('R'+str(row),winner[25].strength_fail)
worksheet.write('S'+str(row),winner[25].strength_check_total)
worksheet.write('T'+str(row),winner[25].drift_x)
worksheet.write('U'+str(row),winner[25].drift_y)
worksheet.write('V'+str(row),winner[25].drift_check)
worksheet.write('W'+str(row),winner[25].weight)
worksheet.write('X'+str(row),winner[25].p1)
worksheet.write('Y'+str(row),winner[25].p2)
worksheet.write('Z'+str(row),winner[25].p3)
worksheet.write('AA'+str(row),winner[25].p4)
worksheet.write('AB'+str(row),winner[25].p5)
worksheet.write('AC'+str(row),winner[25].p6)
worksheet.write('AD'+str(row),winner[25].constraint)
worksheet.write('AD'+str(row),winner[25].num_UL_Pu)
worksheet.write('AE'+str(row),winner[25].num_UL_Mu)
worksheet.write('AF'+str(row),winner[25].num_SVS_Pu)
worksheet.write('AG'+str(row),winner[25].num_SVS_Mu)
print winner[25].num_UL_Pu,winner[25].num_UL_Mu,winner[25].num_SVS_Pu,winner[25].num_SVS_Mu

evolution_point = 0
for number in range(len(winner)/5):
    if number == 0:
        previous_fit = winner[0].fitness
    elif winner[number*5].fitness < previous_fit:
        evolution_point += 1
    previous_fit = winner[number*5].fitness


worksheet.write('AJ'+str(row), evolution_point )
worksheet.write('AK'+str(row), top_10_fit_list[0])
worksheet.write('AL'+str(row), top_10_fit_list[1])
worksheet.write('AM'+str(row), top_10_fit_list[2])
worksheet.write('AN'+str(row), top_10_fit_list[3])
worksheet.write('AO'+str(row), top_10_fit_list[4])
worksheet.write('AP'+str(row), top_10_fit_list[5])

worksheet.write('AR'+str(row), top_30_fit_list[0])
worksheet.write('AS'+str(row), top_30_fit_list[1])
worksheet.write('AT'+str(row), top_30_fit_list[2])
worksheet.write('AU'+str(row), top_30_fit_list[3])
worksheet.write('AV'+str(row), top_30_fit_list[4])
worksheet.write('AW'+str(row), top_30_fit_list[5])

worksheet.write('AY'+str(row), all_fit_list[0])
worksheet.write('AZ'+str(row), all_fit_list[1])
worksheet.write('BA'+str(row), all_fit_list[2])
worksheet.write('BB'+str(row), all_fit_list[3])
worksheet.write('BC'+str(row), all_fit_list[4])
worksheet.write('BD'+str(row), all_fit_list[5])

worksheet.write('BF'+str(row), top_10_mat_list[0])
worksheet.write('BG'+str(row), top_10_mat_list[1])
worksheet.write('BH'+str(row), top_10_mat_list[2])
worksheet.write('BI'+str(row), top_10_mat_list[3])
worksheet.write('BJ'+str(row), top_10_mat_list[4])
worksheet.write('BK'+str(row), top_10_mat_list[5])

worksheet.write('BM'+str(row), top_30_mat_list[0])
worksheet.write('BN'+str(row), top_30_mat_list[1])
worksheet.write('BO'+str(row), top_30_mat_list[2])
worksheet.write('BP'+str(row), top_30_mat_list[3])
worksheet.write('BQ'+str(row), top_30_mat_list[4])
worksheet.write('BR'+str(row), top_30_mat_list[5])

worksheet.write('BT'+str(row), top_5_mat_list[0])
worksheet.write('BU'+str(row), top_5_mat_list[1])
worksheet.write('BV'+str(row), top_5_mat_list[2])
worksheet.write('BW'+str(row), top_5_mat_list[3])
worksheet.write('BX'+str(row), top_5_mat_list[4])
worksheet.write('BY'+str(row), top_5_mat_list[5])
```

```
    worksheet.write('CA'+str(row), all_mat_list[0])
    worksheet.write('CB'+str(row), all_mat_list[1])
    worksheet.write('CC'+str(row), all_mat_list[2])
    worksheet.write('CD'+str(row), all_mat_list[3])
    worksheet.write('CE'+str(row), all_mat_list[4])
    worksheet.write('CF'+str(row), all_mat_list[5])

    plt.close()

# Create an new Excel file and add a worksheet.
workbook = xlsxwriter.Workbook('0919_8at10by2.xlsx')
worksheet = workbook.add_worksheet()

# Widen the first column to make the text clearer.
#worksheet.set_column('A:A', 20)

# Add a bold format to use to highlight cells.
bold = workbook.add_format({'bold': True})

# Write some simple text.

#for the diversity filer
worksheet.write('B1', 'Layout', bold) #sig_cs  1-3
worksheet.write('C1', 'Material_factor', bold) #sig_cs  1-3
worksheet.write('D1', 'Rate_min_ele', bold)  #rate_min_ele
worksheet.write('E1', 'Rate_min_ele', bold)  #rate_lowerbound_ele
worksheet.write('F1', 'Max_distance', bold)  #rate_min_ele
worksheet.write('G1', 'Slop_torsion', bold)  #rate_lowerbound_ele
worksheet.write('H1', 'Preferred_distance', bold)  #rate_lowerbound_ele
worksheet.write('I1', 'Slop_distr', bold)  #rate_lowerbound_ele
worksheet.write('J1', 'Rate_max_distr', bold)  #rate_lowerbound_ele
worksheet.write('L1', 'I_total_x for lay1', bold)
worksheet.write('M1', 'Icr_x for lay1', bold)
worksheet.write('N1', 'Icr_x/I_total_x', bold)
worksheet.write('O1', 'I_total_y for lay1', bold)
worksheet.write('P1', 'Icr_y for lay1', bold)
worksheet.write('Q1', 'Icr_y/I_total_y', bold)
worksheet.write('R1', 'Times of strength failure', bold)
worksheet.write('S1', 'Strength check', bold)
worksheet.write('T1', 'Drift_x', bold)
worksheet.write('U1', 'Drift_y', bold)
worksheet.write('V1', 'Drift check', bold)
worksheet.write('W1', 'Material', bold)
worksheet.write('X1', 'C_Torsion', bold)
worksheet.write('Y1', 'C_Shear', bold)
worksheet.write('Z1', 'C_Opening', bold)
worksheet.write('AA1', 'C_Flexure', bold)
worksheet.write('AB1', 'C_Drfit', bold)
worksheet.write('AC1', 'C_Distribution', bold)
worksheet.write('AD1', 'UL Pu Fail', bold)
worksheet.write('AE1', 'UL Mu Fail', bold)
worksheet.write('AF1', 'Service Pu Fail', bold)
worksheet.write('AG1', 'Service Mu Fail', bold)
worksheet.write('AH1', 'Distance_cm_cf', bold)

### Give a matrix from Latin-Hypercube
#lhs_matrix = lhs(9, samples=N, criterion='center')
dict_lhs_matrix = lhs(4, samples=N, criterion='center')

### the change of parameters:

for figure_i in range(N):


    if dict_lhs_matrix[figure_i][0]< 1.0/3:
        constraint = 1000
    if dict_lhs_matrix[figure_i][0]>= 1.0/3 and dict_lhs_matrix[figure_i][0]<= 2.0/3:
        constraint = 10000
    if dict_lhs_matrix[figure_i][0] > 2.0/3:
        constraint = 100000
    weight_sig = round((1+dict_lhs_matrix[figure_i][1]*2)) ###1~3
    torsion_sig = round((1+dict_lhs_matrix[figure_i][2]*1))     ###1~2
    distribution_sig = round((1+dict_lhs_matrix[figure_i][3]*2))    ### 1~3
    preferred_distance = 5 #round((0.5+dict_lhs_matrix[figure_i][4]*2.5)*10)/10.0  #0.5-3 ft
    rate_max_distribution = 0.667 #round((0.5+dict_lhs_matrix[figure_i][7]*0.5)*10)/10.0  #0.5-1.0
    acceptable_distribution_x = rate_max_distribution*dimension_x/2
    acceptable_distribution_y = rate_max_distribution*dimension_y/2
    # Write the parameters in each row.
    row = figure_i+2
    main_loop(figure_i+1, constraint,weight_sig,torsion_sig,distribution_sig)

workbook.close()

# end of the code
```

Bibliography

[1] *ACI 318-14 Building Code Requirements for Structural Concrete and Commentary.* American Concrete Institute.

[2] Hosseini Mahmood Aminnia Mohammad. The effects of placement and crosssection shape of shear walls in multistory rc buildings with plan irregularity on their seismic behavior by using nonlinear time history analyses. *Int J Civil Environ Struct Constr Archit Eng*, 9:1293300, 2015.

[3] Wong Kin-Ming Chan Chun-Man. Structural topology and element sizing design optimisation of tall steel frameworks using a hybrid ocga method. *Struct Multidiscip Optim*, 35:47388, May 2008.

[4] Greenberg H Dorn W, Gomory R. Automatic design of optimal structures. *J Mec*, 3:25–51, 1964.

[5] Sokó et al. On the adaptive ground structure approach for multi-load truss topology optimization. *In: 10th World Congress on Structural and Multidisciplinary Optimization, Orlando, Florida, USA*, 2013.

[6] Stiny George. Shape: Talking about seeing and doing. *The MIT Press*, 2013.

[7] Lai H-J. Contactions and hamiltonian line graphs. *J Graph Theory*, 12:115, 1988.

[8] Ohsaki M Hagishita T. Topology optimization of trusses by growing ground structure method. *Struct Multidisc Optim*, 37:377–93, 2009.

[9] Lawrence C Kamara Mohmoud E. *Simplified Design of Reinforced Concrete Buildings.* Portland Cement Association, 2011.

[10] Shekhawat Krishnendra. Algorithm for constructing an optimally connected rectangular floor plan. *Front Archit Res*, 3:32430, 2014.

[11] Steven Grant P Liang Qing-Quan, Xie Yi-Min. Optimal topology design of bracing systems for multistory steel frames. *J Struct Eng*, 126(7), 2000.

[12] Mitchell Melanie. *An Introduction to Genetic Algorithms.* The MIT Press, Cambridge, Massachusetts; London, England, 1998.

[13] Charles W Nilson Arthur H, Darwin David. *Design of concrete structures.* Dolan: McGraw Hill Higher Education, 2003.

[14] Bao Fan Fink Daniel Yan Dong-Ming Wonka Peter et al Peng Chi-Han, Yang Yong-Liang. Computational network design from functional specifications. *ACM Trans Graphics*, 2016.

[15] Kostas Terzidis. Autoplan: a stochastic generator of architectural plans from a building program. *Digital Media and the Creative Process*, 2007.