

OPTIMISED DESIGN OF HIGH VOLTAGE LATTICE TRANSMISSION TOWERS

JOSÉ DIOGO PEREIRA FILGUEIRAS DA MOTA

Dissertação submetida para satisfação parcial dos requisitos do grau de
MESTRE EM ENGENHARIA CIVIL — ESPECIALIZAÇÃO EM ESTRUTURAS

Orientador: Professor Doutor José Miguel de Feitas Castro

Coorientador: Engenheiro Luís Augusto Rodrigues de Macedo

JUNHO DE 2017

MESTRADO INTEGRADO EM ENGENHARIA CIVIL 2016/2017

DEPARTAMENTO DE ENGENHARIA CIVIL

Tel. +351-22-508 1901

Fax +351-22-508 1446

✉ miec@fe.up.pt

Editado por

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Rua Dr. Roberto Frias

4200-465 PORTO

Portugal

Tel. +351-22-508 1400

Fax +351-22-508 1440

✉ feup@fe.up.pt

🌐 <http://www.fe.up.pt>

Reproduções parciais deste documento serão autorizadas na condição que seja mencionado o Autor e feita referência a *Mestrado Integrado em Engenharia Civil - 2016/2017 - Departamento de Engenharia Civil, Faculdade de Engenharia da Universidade do Porto, Porto, Portugal, 2017.*

As opiniões e informações incluídas neste documento representam unicamente o ponto de vista do respetivo Autor, não podendo o Editor aceitar qualquer responsabilidade legal ou outra em relação a erros ou omissões que possam existir.

Este documento foi produzido a partir de versão eletrónica fornecida pelo respetivo Autor.

A meus Pais e irmã

*In times of change, learners inherit the earth
while the learned find themselves beautifully
equipped for a world that no longer exists.*

Eric Hoffer

ACKNOWLEDGEMENTS

I would like to express my gratitude to all those who accompanied me during this important part of my life.

To my family for shaping me into the persistent, dedicated and creative person I am today, traits that helped me to pursue this unconventional theme.

To all my good friends, for the fun, the good moments and for the unforgettable memories we shared during the last five years.

To Professor José Miguel Castro for accompanying me, first as a professor and later as the supervisor of this dissertation. For sharing this passion for technology and for providing me with useful technical and personal advice.

To Engineer Luis Macedo for the knowledgeable insights provided about Machine Learning and optimization algorithms.

To the industry partners at Metalogalva, Engineers Rui Cunha and Hélder Costa, for the useful technical input provided and interest demonstrated.

To my mentor, James Parker, and everyone at Expedition Engineering, where I worked with extremely talented engineers, and got immersed in an environment that encouraged innovation and creativity that ultimately helped me arrive at the theme of this dissertation.

ABSTRACT

Emergent technologies are expanding into other sectors, often causing disruption in the way companies operate. The construction industry is a very good target for this expansion of technology given the potential productivity gains to be expected.

A recent study by McKinsey, concluded that, over the last 20 years, labour productivity in the construction sector has lagged behind overall economic productivity. R&D spending is less than 1% of revenues in the sector and it is one of the least digitized industries. On the other hand, an estimated \$57 trillion are expected to be spent on infrastructure by 2030, the report ends up concluding that the construction industry is “ripe for disruption”.

These new technologies will disrupt traditional workflows and, in the medium term, the role of the engineer will change. Code compliance checks will be mostly automated leaving the engineer free to do more creative work, and that role will often be augmented by computational design tools.

Computational design is one of the fields that is starting to grow in importance in design practices. Currently, most of these tools are used in the conceptual design stage, where full code compliance is not pursued. However, in the final project – where code compliance is critical – such tools have their importance reduced because of the lack of compatibility of the automatically generated solutions with the design codes.

The present work studies the possibility of creating a design tool that can be used not only in the conceptual phase but also in the final stage where a code compliant structure is needed.

The structures optimized in this dissertation are high voltage electricity pylons. This choice was made as the design process is simple enough to be developed in the time frame of the present work, in comparison to other types of structures (buildings, high rise towers, bridges, etc). Although simple in the design phase, the current design codes for such structures are quite strict when it comes to geometry. This provides the main challenge of this thesis, which is to develop an algorithm known to produce organic shapes and make it work according to current design codes.

Valuable technical insight was gathered from Metalgalva – industry partner – that also provided a case study model that was used to benchmark the developed tool against a conventional lattice tower structure.

The result was a software tool capable of returning code compliant lattice towers with 10 % material savings in comparison to the case study model. During the dissertation, several additional modules needed to create code friendly genetic algorithms were also identified. Such findings can be useful for future work dedicated to this field of code compliant computational design tools.

KEYWORDS: Genetic Algorithm, Optimization, Steel Tower, Lattice, Electricity Pylon.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	i
ABSTRACT	iii
1. INTRODUCTION	1
1.1. OBJECTIVES	1
1.2. ELECTRICITY PYLONS	2
1.3. STRUCTURE	3
2. DESIGN OF LATTICE TOWERS	5
2.1. GENERAL PRINCIPLES	5
2.2. DESIGN LOADS	6
2.3. RESISTANCE VERIFICATION	7
2.3.1. IMPLEMENTATION NOTES	7
2.4. STABILITY VERIFICATION	7
2.4.1. EFFECTIVE SLENDERNESS FACTOR - K	7
2.4.1.1. Implementation notes – Leg members	8
2.4.1.2. Implementation notes – Diagonal bracing members	9
2.4.1.3. Implementation notes – Horizontal bracing members	10
2.4.2. BUCKLING LENGTH	10
2.5. CONNECTION DESIGN	14
3. GENETIC ALGORITHM	15
3.1. OVERVIEW	15
3.2. FUNDAMENTAL ELEMENTS OF A GENETIC ALGORITHM	16
3.2.1. INDIVIDUAL	16
3.2.2. INITIAL POPULATION	17
3.2.3. FITNESS FUNCTION	18
3.2.4. SELECTION FUNCTION	19
3.2.4.1. Roulette wheel	19
3.2.4.2. Tournament selection	20
3.2.5. GENETIC OPERATORS	20

3.2.5.1. Crossover	20
3.2.5.2. Mutation	21
3.2.6. TERMINATION CRITERIA	21
4. PROGRAM DESCRIPTION	23
4.1. SUMMARY	23
4.2. PROGRAM FLOW	23
4.3. BASE GENETIC CODE	25
4.4. INITIAL POPULATION	29
4.5. EVALUATE FUNCTION	30
4.6. REPAIR FUNCTION	33
4.7. SELECTION FUNCTION	35
4.8. GENETIC OPERATORS	35
4.8.1. CROSSOVER	35
4.8.2. GAUSSIAN MUTATION	36
5. CASE STUDY	39
5.1. BASE MODEL	39
5.2. PROGRAM SETUP	41
5.2.1. TEST RUNS AND CALIBRATION	42
5.3. RESULTS	44
5.3.1. POSTPROCESSING	44
5.3.2. ANALYSIS	46
5.4. CONCLUDING REMARKS	47
5.3.2. FUTURE WORK	47
REFERENCES	49

LIST OF FIGURES

Fig.1.1 – Power lines designed to withstand snow, Iceland.....	2
Fig.1.2 – Test bench – India.....	3
Fig.2.1 – Steel poles.....	6
Fig.2.2 – Lattice tower	6
Fig.2.3 – EC3-3-1 table G.1 from annex G	8
Fig.2.4 – EC3-3-1 table G.2 from annex G	9
Fig.2.5 – EC3-3-1 table G.3 from annex G	10
Fig.2.6 – EC3-3-1 Figure H.1 from annex H	11
Fig.2.7 – Increased number of horizontal divisions.....	11
Fig.2.8 – EC3-3-1 Figure H.3 from annex H	12
Fig.2.9 – EC3-3-1 Figure H.4 from annex H	13
Fig.3.1 – Flow of standard GA.....	15
Fig.3.2 – Chromosome structure and values	16
Fig.3.3 – Basic bit string chromosome	17
Fig.3.4 – More complex chromosome datatype	17
Fig.3.5 – Random vs Seeded population	17
Fig.3.6 – Roulette wheel: A is better than G, therefore selected more often	19
Fig.3.7 – Tournament selection.....	20
Fig.3.8 – Double and single point crossover	21
Fig.4.1 – User interface	24
Fig.4.2 – Interaction between key modules.....	25
Fig.4.3 – Grasshopper.....	26
Fig.4.4 – Grasshopper and rhino model.....	27
Fig.4.5 – IAM: Different bracing conditions	31
Fig.4.6 – Tapered normal distribution	36
Fig.5.1 – Base model	39
Fig.5.2 – Arm nodes	40
Fig.5.3 – Mutation adjustment trial	43
Fig.5.4 – Ideal mutation probability reached	43
Fig.5.5 – Final search.....	43
Fig.5.6 – Plan view of the tower	44

Fig.5.7 – Critical quadrant, front and side planes.....	45
Fig.5.8 – Legs with secondary bracing added.....	45

TABLE LIST

Table 4.1 – DNA definition	28
Table 4.2 – Initial mutation implementation.....	29
Table 4.3 – Evaluate function call	30
Table 4.4 – IAM scan (leg scan excerpt).....	30
Table 4.5 – EC3 verification for leg members	32
Table 4.6 –Repair function excerpt	34
Table 4.7 – Tournament selection.....	35
Table 4.8 – Crossover function applied to the bar array	35
Table 4.9 – Gaussian mutation algorithm	37
Table 4.10 – Clamp function	37
Table 4.11 – Mutation function	37
Table 5.1 – Base model weight distribution	40
Table 5.2 – Load cases	41
Table 5.3 – Final structure weight distribution.....	46

1

Introduction

Improvements in computational power are supporting the growth of new fields such as Machine learning, artificial intelligence, and data science. The potential contained in these fields for increased productivity and efficiency, is allowing technology to break into sectors that have been in the past very reticent to change and progress. This stage is commonly designated by the 4th Industrial Revolution.

A good example of the slow adoption of innovative technologies is the construction industry, a sector that contributes massively to several pollution indicators such as air quality, emission of climate change gasses, and ozone depletion.

Over the last two decades, talks in the construction industry about new methods and solutions have been increasing, but when compared to other industries, actual implementation of these new methods has been slow. Currently, the technology is making its way into the design stage, with technologies such as Building Information Modelling (BIM) already widely used, mandatory in some projects, and with technologies such as Virtual and Augmented Reality and fields of Algorithmic and Computational design rapidly expanding in design practices.

Computational and Algorithmic design have the potential to become the most disruptive change in the design workflow. Currently, they work very well at the conceptual design phase and have reduced value later in the design process as they clash with the often overly conservative design codes that are written in a way that rewards the traditional solutions and leaves little chance for innovative approaches to a design problem.

1.1. OBJECTIVES

This dissertation assesses the feasibility of implementing these new emerging fields past the concept design phase into the final project. The structures optimized in this dissertation are high voltage electricity pylons. This choice was made as the design process is simple enough to be developed in the time frame of the present work, when compared to other options (buildings, high rise towers, bridges, etc). Although simple in the design phase, the current design codes for such structures are quite strict when it comes to geometry. This provides the main challenge of this thesis, which is to develop an algorithm known to produce organic shapes and make it work according to current design codes.

If this integration proves possible, the result will be a software tool, with potential commercial implementation, that outputs an optimized structure, reducing material usage and design time. The engineer's function will be reduced to the initial input of information, supervise the algorithm and later design and detailing of connections in the structure. Overall, the engineer will be freed to do more creative work.

If the integration is unsuccessful, the present work will at least provide insights about this often-conflicting relationship between current design codes and future technology, useful in the inevitable future research conducted in this field.

1.2. ELECTRICITY PYLONS

To transmit electricity over large distances, several solutions are used, from the electricity pylons to underground cables in locations where the visual disruption of tower structures is unacceptable or the risk of power outages in wind/snow storms justifies the extra costs of this solution.

The first electricity pylons were erected in the early 1920s, initially made of wood. They have rapidly evolved to the traditional steel design we see today. Since that material change, the fundamental design principles of such structures remained largely unchanged.

The lattice solution is the most commonly used. There are several reasons for the adoption of this solution. Lattices can be applied to very small or very large towers and are often the cheapest solution when other constraints such as visual impact are not present. They can vary in shape to withstand different load conditions (Figure 1.1).



Fig. 1.1 – Power lines in Iceland designed to withstand snow

They are usually designed to make use of economies of scale. For that reason, a pylon is designed to withstand a wide range of loads present in the initial project of a new power line. This ensures that the same pylon can be used several times along the power line, reducing material waste and design time. As pylons are replicated several times, and have a relatively small size, they are one of the few structures in this field that can be tested before deployment to validate and improve the design. Such tests are conducted in bespoke test benches (Figure 1.2).



Fig. 1.2 – Test bench – India

There are also other tower solutions to transmit electricity such as very high strength steel poles with circular hollow sections. They have a reduced visual impact, are assembled faster and the base occupies less space. However, they are more expensive to build and maintain and are prone to dynamic effects due to wind loads.

1.3. STRUCTURE

This dissertation is divided in 5 chapters. After this initial introduction, Chapter 2 details the design requirements prescribed in the European standards, namely EN 50341-1, EN 1993-1-1 and EN 1993-3-1, that the lattice towers need to meet. The details of design assumptions that needed to be made to make possible the development of a program with a high degree of automation are also discussed.

Chapter 3 starts by giving an overview of what a genetic algorithm is and proceeds to explain the key components present in these types of optimisation algorithms.

Chapter 4 presents the various components of the application developed. Every component is described in detail along with the presentation of some excerpts of code.

In Chapter 5 the results of a case study are presented and analysed.

2

Design of lattice towers

In this chapter, design verifications required for this type of structure are detailed. The tower structures in the present work are lattice towers with angle sections and are designed according to EN 50341-1. Resistance checks are done according to EN 1993-1-1 (EC3-1-1) and stability calculations follow Annex G and H of the EN 1993-3-1 (EC3-3-1).

To develop a program with a high level of automation, some design assumptions were required. When relevant, such assumptions and other implementation details will be detailed in the following subchapters.

2.1. GENERAL PRINCIPLES

According to clause 7.3.5 of EN 50341-1, a lattice tower should be checked based on the results obtained from a global elastic analysis. The model is usually pin jointed but if the continuity between members is considered the bending moments can be neglected.

The code identifies three types of elements: leg members, bracing and secondary bracing elements. As the secondary bracing elements – also called redundant members – do not receive direct loads and are in place just to assure local stability of the load bearing members, they can be ignored in the global analysis.

Sections are classified according to EC3 and, when class 4, the effective area should be used in the design verifications. The effective area is calculated according to EC3 unless the sections are hot rolled angles. For that case, clause 7.3.6.2 of EN 50341-1 provides the following expressions to calculate the effective area:

$$\rho = 1 \text{ if } \bar{\lambda}_p \leq 0.748 \quad (2.1)$$

$$\rho = \frac{\bar{\lambda}_p - 0.188}{\bar{\lambda}_p^2} \leq 1 \text{ if } \bar{\lambda}_p > 0.748 \quad (2.2)$$

$$\bar{\lambda}_p = \frac{(h - 2t)/t}{28.4\epsilon\sqrt{K\sigma}} \text{ or } \bar{\lambda}_p = \frac{(b - 2t)/t}{28.4\epsilon\sqrt{K\sigma}} \quad (2.3)$$

Deformations and vibrations under SLS loads and fatigue are not normally considered unless specified on the project.

2.2. DESIGN LOADS

Lattice tower structures, as any type of structure, are subject to various loads during their life time. The present work will not go into detail on every combination, instead the critical loads will be presented along with other relevant information provided by Metalgalva.

Lattice towers used on energy transmission lines have their design loads defined according to EN 50341, along with the self-weight of the structure itself, the tower must be able to withstand the loads applied by the cables.

The actions of the cables are applied to the structure as a point load at the end of each arm. The critical loads when designing this type of structure are:

- Self-weight of the cables
- Wind loads on the cables
- Force from change of direction on the transmission line (in angle towers)
- Force due to cable failure
- Anchorage force on terminal towers

Seismic and fire actions are usually not considered unless specified by the client. Wind loads in the structure are also taken into account when designing these types of structures. According to EN 50341, the bending moments created by this action on individual elements can be ignored. Wind loads are usually critical for steel poles (Figure 2.1), however that is not observed in lattice structures (Figure 2.2), due to its shape, normally wind loads add just 10% more stresses on the structural elements.



Fig. 2.1 – Steel Poles



Fig. 2.2 – Lattice tower

2.3. RESISTANCE VERIFICATION

For this type of structure, the members should be checked according to EC3 for both tension and compression elements. As this is a pin jointed structure, only axial loads are considered, as such, the expression used is.

$$N_{rd} = \frac{A * f_y}{\gamma_{M0}} \text{ with } A = A_{eff} \text{ for class 4 sections} \quad (2.4)$$

2.3.1. IMPLEMENTATION NOTES

There is a provision in Eurocode 3 regarding angles connected through one leg. Since the software outputs a structure where the connections are not yet designed this verification is not implemented in the final program.

The area used by the program to calculate N_{rd} is the one received by the user. For that reason, a previous class classification is required for each cross-section to reduce the area for the case of a class 4 cross-section.

2.4. STABILITY VERIFICATION

Buckling resistance of members in compression was checked using annex G and H of EC3-3-1.

The expression used to determine the buckling resistance ($N_{b,rd}$) of a member is available on Part 1 of EC3-1-1:

$$N_{b,Rd} = \frac{\chi A f_y}{\gamma_{M0}}, A = A_{eff} \text{ for class 4 sections} \quad (2.5)$$

The steps needed to determine the reduction factor χ and factor Φ are detailed in Annex G of the EC3-3-1. A new effective slenderness ratio, $\overline{\lambda}_{eff}$, should be used to calculate both factors, this new slenderness ratio is determined by multiplying the original slenderness ratio (from Eurocode 3) by a factor K that varies with the member being calculated (leg, diagonal or horizontal bracing member).

$$\overline{\lambda}_{eff} = K \bar{\lambda} \text{ with } \bar{\lambda} = \frac{\lambda}{\lambda_1} \quad (2.6)$$

To determine λ , Annex H is used to get the buckling length relevant to the member and λ_1 is calculated according to EC3 part 1.

The effective slenderness factor, K, is calculated from the expressions on tables G.1 to G.3 from the Eurocode 3 that list various values depending on the section type, buckling axis, geometry and member type.

2.4.1. EFFECTIVE SLENDERNESS FACTOR - K

The effective slenderness factor, is used to reflect elements with different likelihoods of instability in the lattice tower, as such, a distinction is made between element types when k values are assigned to each bar.

Leg members have their k value assigned using table G.1. For diagonal bracing elements, k should be determined taking into account both the bracing pattern and the connections of the bracing legs. In the absence of more accurate values, k should be obtained from table G.2. Finally, for the horizontal bracing members table G.1 is used and it can be combined with an extra factor calculated using table G.3 for certain bracing configurations.

These tables are presented in the next subchapters detailing the implementation in the algorithm.

2.4.1.1. Implementation notes – Leg members

For leg members, as the software is intended to design only with single angle elements and given the characteristics of the base structure (detailed in chapter 4), geometry of the type described as case (d) in Table G.1 is not possible, as such, the expression used to calculate the effective slenderness factor, K, is given by:

$$k = 0.8 + \frac{\bar{\lambda}}{10} \text{ but } k > 0.9 \text{ and } k < 1.0 \quad (2.7)$$

Even though case (d) – “discontinuous top end with horizontals” – is not allowed as a possible solution case (e) is possible. This case uses the same expression to determine the effective slenderness factor, K, according to another axis, y-y. As the algorithm does not know if the structural solution is symmetrical or unsymmetrical there is a function (detailed in chapter 4) that builds the internal analytical model (IAM) and states the axis in which stability checks need to be carried on Leg members.


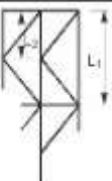
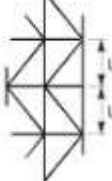
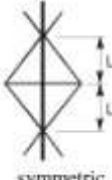
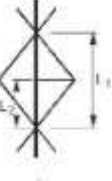

Symmetrical bracing			Unsymmetrical bracing			
Section	L ⁽¹⁾		Section	L ⁽²⁾		
Axis	v-v	y-y	Axis	v-v	y-y	y-y
 Case (a) Primary bracing at both ends	$0.8 + \frac{\bar{\lambda}}{10}$ but ≥ 0.9 and ≤ 1.0	1.0 ⁽¹⁾	 discontinuous top end with horizontals Case (d) Primary bracing at both ends	$1.2 \left(0.8 + \frac{\bar{\lambda}}{10} \right)$ but ≥ 1.08 and ≤ 1.2 on $I_{z^{(2)}}$	$1.2 \left(0.8 + \frac{\bar{\lambda}}{10} \right)$ but ≥ 1.08 and ≤ 1.2 on I_{z1}	1.0 on I_{z1} ⁽¹⁾
 asymetric  symmetric Case (b) Primary bracing at one end and secondary bracing at the other	$0.8 + \frac{\bar{\lambda}}{10}$ but ≥ 0.9 and ≤ 1.0	1.0 ⁽¹⁾	 Case (e) Primary bracing at both ends	$0.8 + \frac{\bar{\lambda}}{10}$ but ≥ 0.9 and ≤ 1.0 on $I_{z^{(2)}}$	$0.8 + \frac{\bar{\lambda}}{10}$ but ≥ 0.9 and ≤ 1.0 on I_{z1}	1.0 on I_{z1} ⁽¹⁾
 Case (c) Secondary bracing at both ends	$0.8 + \frac{\bar{\lambda}}{10}$ but ≥ 0.9 and ≤ 1.0	1.0				

Fig. 2.3 – EC3-3-1 table G.1 from Annex G

2.4.1.2. Implementation notes – Diagonal bracing members

In a genetic algorithm that automatically iterates through various topologies it is not possible to analyse, without significant computation, the spatial relationship between all the members in each plane with sufficient detail to determine a bracing pattern, in fact it is highly likely that some solutions of the GA do not fit a predefined bracing pattern listed in Annex H. Having this into account, the more precise definition of a k value is not possible and, for that reason, as stated in annex G, for this situation, the “worst case” should be adopted and the k value obtained from Table G.2.

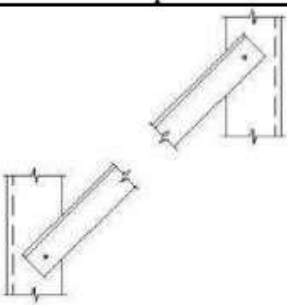
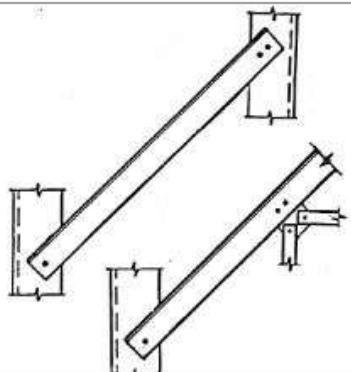
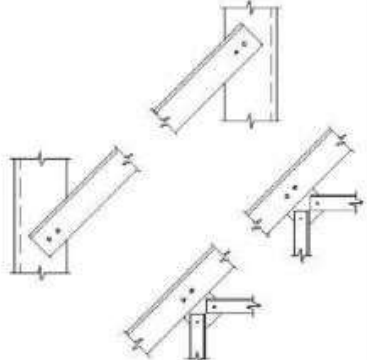
Type of restraint	Examples	Axis	k
Discontinuous both end (i.e. single bolted at both ends of member)		v-v	$0,7 + \frac{0,35}{\lambda_v}$
		y-y	$0,7 + \frac{0,58}{\lambda_y}$
		z-z	$0,7 + \frac{0,58}{\lambda_z}$
Continuous one end (i.e. single bolted at one end and either double bolted or continuous at other end of member)		v-v	$0,7 + \frac{0,35}{\lambda_v}$
		y-y	$0,7 + \frac{0,40}{\lambda_y}$
		z-z	$0,7 + \frac{0,40}{\lambda_z}$
Continuous both ends (i.e. double bolted at both ends, double bolted at one end and continuous at other end, or continuous at both ends of the member)		v-v	$0,7 + \frac{0,35}{\lambda_v}$
		y-y	$0,7 + \frac{0,40}{\lambda_y}$
		z-z	$0,7 + \frac{0,40}{\lambda_z}$

Fig. 2.4 – EC3-3-1 Table G.2 from Annex G

To extract values for K from Table G.2, additional information regarding connection types for bracing members is needed, as previously stated, the software developed for this thesis is intended to be used in the definition of the tower geometry, as such, the connection types are not yet known. In this case an option was made to work with the expressions for the worst-case scenario – higher k values – where bracing members are discontinuous and single bolted at both ends.

2.4.1.3. Implementation notes – Horizontal bracing members

Horizontal bracing members have the same rules of diagonal members with the exception of K bracing patterns where the K value needs to be reduced by a ratio determined from table G.3 to account for a member with both tension and compression in each half of its length.

Ratio $\frac{N_t}{N_c}$	Modification factor, k_1
0,0	0,73
0,2	0,67
0,4	0,62
0,6	0,57
0,8	0,53
1,0	0,50

Fig. 2.5 – EC3-3-1 Table G.3 from Annex G

2.4.2. BUCKLING LENGTH

To determine the buckling length of the members, EC3 does the same distinction between leg, diagonal and horizontal bracing members. Annex H details the steps to take for each type of element.

For leg members, as the algorithm was developed for regular towers where single angle sections were used for the bars and perpendicular truss planes, clause 2 for H.2 is met and the buckling length for leg members is the distance between two nodes.

Given the type of structural model used there is a possibility that an element in the final structure is made of several bar elements in the structural analysis model. To take this detail into account an auxiliary algorithm was used to create the internal analytical model (IAM) that assembles several bars into one when certain conditions are met. This algorithm is described in detail in Chapter 4.

Diagonal bracing elements are differentiated between primary and secondary elements, to develop a program that finds the best solution with minimal user interactions a trade-off was made: every bracing member starts as a primary bracing element. This requires secondary elements to be evaluated against the stricter criteria that primary elements must pass this obviously makes the sections larger than they have to be.

Metalogalva gave useful technical insight that limited the real-world cost of this trade-off. To make the connection design viable in practical terms there is a minimum section size that can be used, the L40x40x4, with this new practical limitation it became clear that even though the theoretical structure could be lighter the real structure – where connections must be designed – would have no significant weight added from this simplification.

Primary bracing elements have their slenderness calculated with the following expression:

$$\lambda = \frac{L_{di}}{i_{vv}} \text{ for angles} \quad (2.8)$$

The length, L_{di} , is specified from figure H.1 from annex H.

Typical primary spacing patterns					
parallel or tapering			usually tapering		usually parallel
I	II	III	IV	V	VI
Single lattice	Cross bracing	K-bracing	Discontinuous bracing with continuous horizontal intersections	Multiple lattice bracing	Tension bracing
$L_{di} = L_d$	$L_{di} = L_{d2}$	$L_{di} = L_{d2}$	$L_{di} = L_{d2}$		

Fig. 2.6 – EC3-3-1 Figure H.1 from Annex H

As shown in figure 2.6, most buckling lengths are simply the distance between the start and end nodes of bars. There is an exception made for cross bracing, according to H.3.3(1): “Provided that the load is equally split into tension and compression, the members are connected where they cross, and provided also that both members are continuous, the centre of the cross can be considered a point of restraint (...)”.

This point is accounted for not directly in the code that generates the IAM but by allowing the user to set the number of horizontal divisions that the program will test.

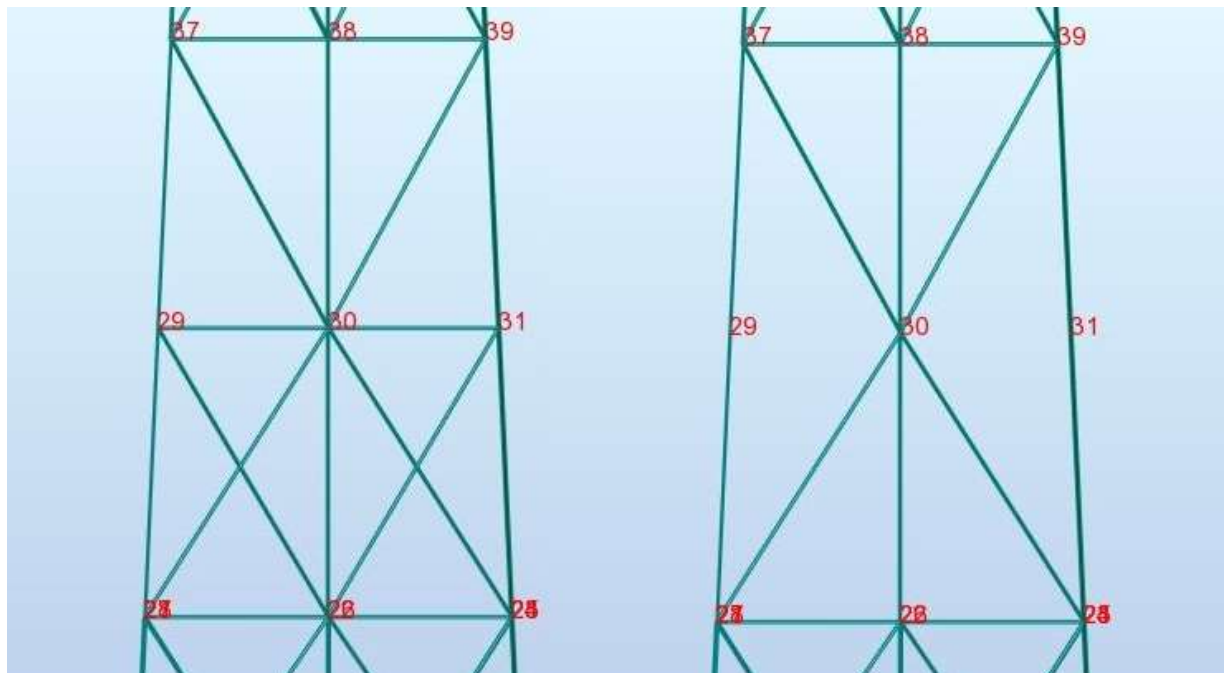


Fig. 2.7 – Increased number of horizontal divisions

As seen in Figure 2.7, increasing the number of horizontal divisions has the same effect as considering the intersection point as a point of restraint, for example the buckling length of the top left diagonal bar remains constant (distance between node 37 – 30) with and without horizontal bars in place. With the evolution of the structure the added horizontal bars will eventually be deleted if the Genetic algorithm deems them unnecessary (as seen on the right of figure 2.7).

Horizontal bracing members need to be checked for buckling in the horizontal plane (transverse stability) and in the frame plane.

To ensure transverse stability for the horizontal members over a certain length, plane bracing should be provided. According to Figure H.3 from EC3 there are two types of plane bracing, triangulated and not fully triangulated.

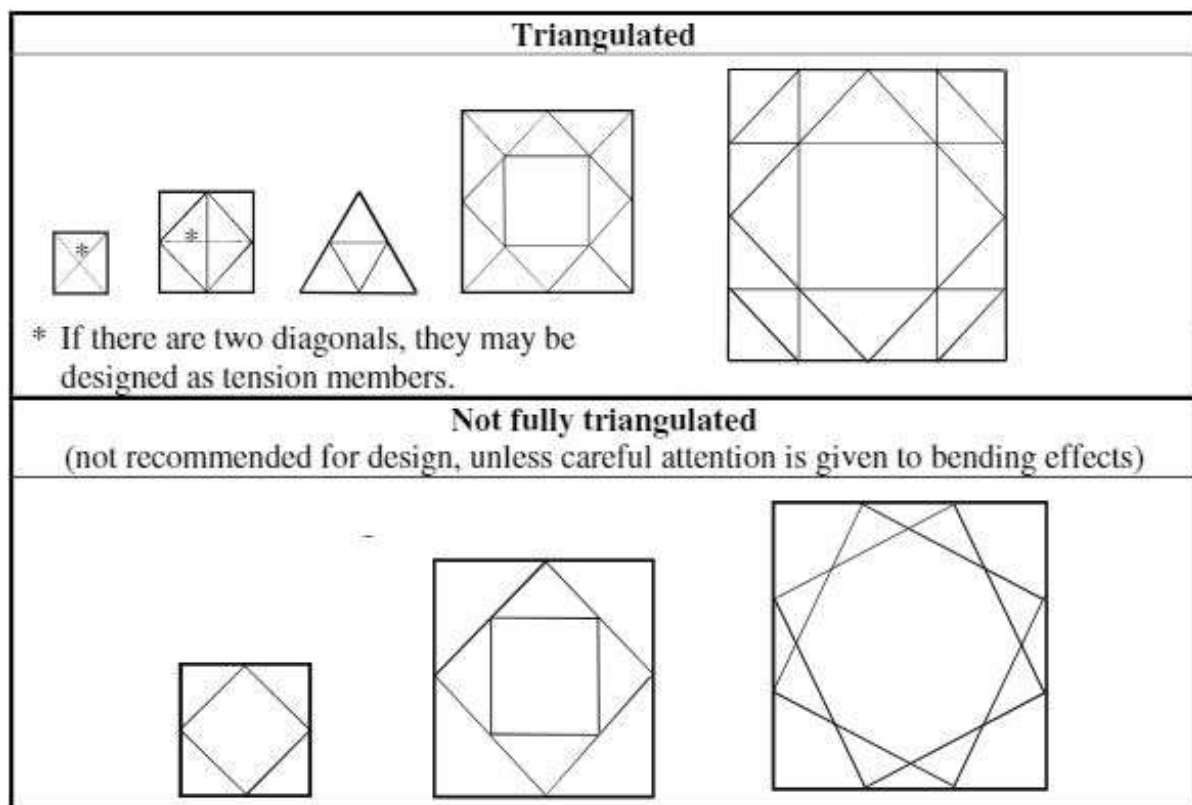


Fig. 2.8 – EC3-3-1 Figure H.3 from Annex H

The algorithm assumes every existing plane bracing to be fully triangulated as it is usually the solution that allows for more material savings in the horizontal bracing members (smaller buckling length). As shown in Figure 2.8, there are several triangulation solutions depending on the degree of subdivision of each side of the tower. Each type of plane bracing would need to be hard coded in the initial structure definition as a function of the subdivision of the tower, although possible, this definition of geometry for each subdivision type would take away too much development time (hard coding several plan bracing geometries) for the objective of the present work, instead an alternative method was found that defined the initial triangulation solution and relied on the user to complete the triangulation. This will obviously be corrected in an eventual commercial spin-off of this application.

According to Annex H of EC3-1-3, the buckling length for transverse stability when there is a need for plane bracing is the distance between intersection points of the horizontal bracing members with the

plane bracing members. When the length is sufficiently small to delete plane bracing the buckling length for single angle members is described in H.3.10 (2) and supported by figure H.4.

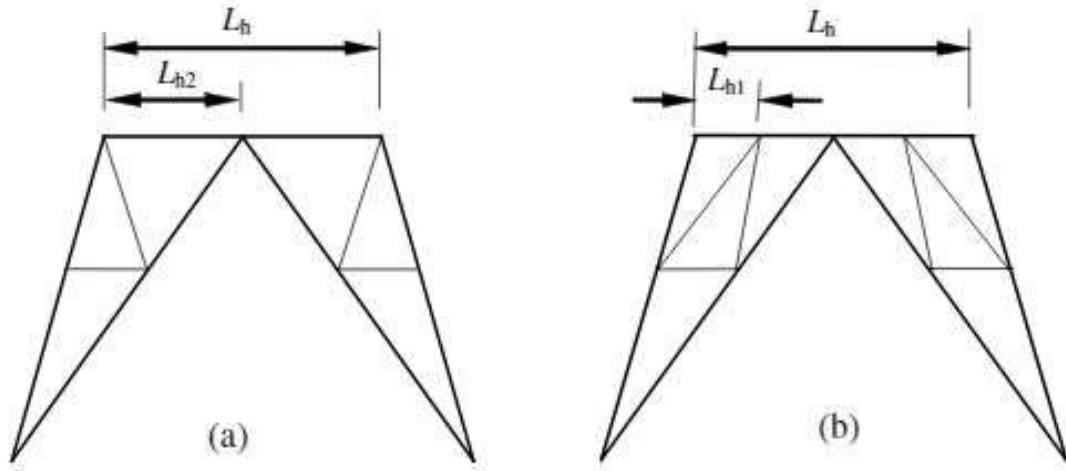


Fig. 2.9 – EC3-3-1 Figure H.4 from Annex H

The length is either half of the total length, L_{h2} , or the distance between intersection points with leg or diagonal bracing members, L_{h1} .

$$\lambda = \frac{L_{h2}}{i_{vv}} \text{ and } \lambda = \frac{L_{h1}}{i_{vv}} \quad (2.9)$$

For stability verification in the plane frame, the buckling length is the distance between intersection points with leg or diagonal bracing members in that plane frame. Figure H.3.9 (3) of EC3-3-1 indicates that, when there is bracing at or near the mid-point of the horizontal bracing members it is possible to use the less restrictive rectangular radius of gyration for the stability verification, when that is not the case the v-v radius of gyration, i_{vv} , should be used. Given the nature of the algorithm it is not known from the start if the structure has bracing members close to the centre, however, tests with several tower sizes have shown that with sufficient initial subdivision levels the probability of horizontal members having no restraint near the centre is very low.

To analyse which development path to take two versions of the same program were created, one using the rectangular gyration axis were allowed and the other using always the i_{vv} . The final results, indicated that, the extra material savings that were to be expected from using the y-y axis were negligible. Most horizontal bars have their section limited by the minimum section required to allow realistic connections to be designed (as detailed previously). The only horizontal bars where savings were obtained were the horizontal bars near the arm elements, that means that the higher the tower, the more diluted these small weight gains become. Based on the results of this test the final version of the program uses the conservative i_{vv} radius of gyration for the stability verification in the plane frame.

2.5. CONNECTION DESIGN

Connection design on this type of structures is done according to Part 1-8 of Eurocode 3. The software developed in this thesis is intended to support the engineer during the geometry definition of the tower. The engineer is then needed to define the connections between elements of the output geometry and sections.

Even though not the objective of the developed application, the software makes the job easier for the engineer by using a minimum section that ensures the connection design stage is completed without changes in the section sizes.

3

Genetic Algorithm**3.1. OVERVIEW**

A genetic algorithm (GA), is an optimisation method inspired in nature and its evolution mechanisms. Starting with a set of typically randomly generated solutions – called individuals of the initial population - the algorithm tries to breed new individuals based on the current generation using operators of selection, crossover and mutation similar to those used by biological organisms.

A GA is a metaheuristic method and, as such, the solution space to be explored is limited by various constraints such as computational capacity, algorithm parameters, problem information and time constraints. Given these limitations the optimum solution returned by the algorithm may not represent the global optimum but a satisfactory solution for the problem. Figure 3.1 shows the flow of a standard GA.

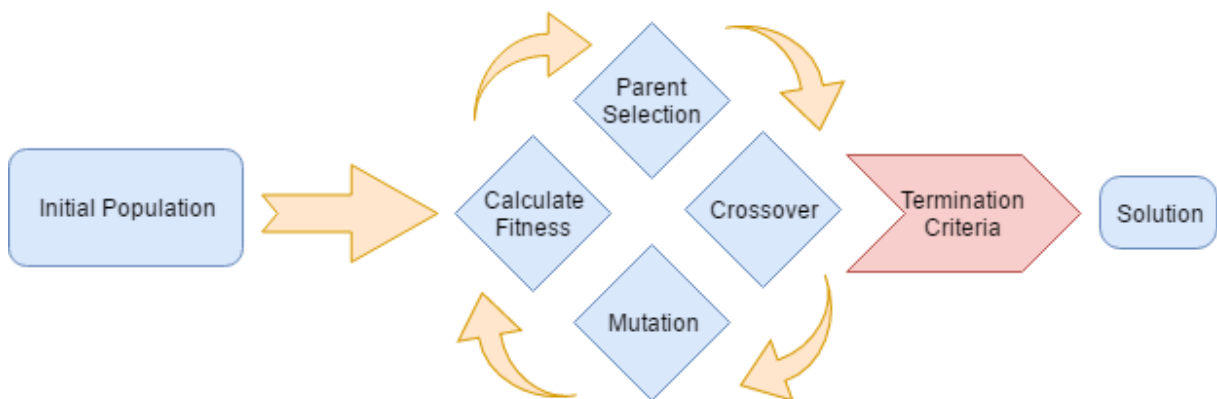


Fig. 3.1 – Flow of a standard GA

The initial population is generated and populated with individuals whose chromosome is randomly generated. Each individual represents a different solution for the problem.

A fitness function runs through every individual and awards a fitness value to each solution. It represents the merit each solution has and is used later to select parents for the next generation. A fitness function can be static or dynamic, i.e. – changing at runtime.

When every individual of the initial population has a fitness value assigned to it, an Evolve command is issued triggering a set of operations. The parents of each new individual are selected by a method

that takes into account the fitness value of each individual of the current population. Individuals with better fitness are more likely to be selected. The selection is passed on to the first genetic operator, usually a crossover function that has the task of assembling a new chromosome made up of blocks from the parents' chromosomes. At this point, the chromosome of the new individual is built, however, to preserve the diversity of solutions during the search of the optimum individual, a mutation operator is used to introduce new information into the chromosome.

The new individual is then evaluated with the fitness function, and the process is repeated until a predefined stop criteria is met. The next subchapters will detail the fundamental elements of a GA.

3.2. FUNDAMENTAL ELEMENTS OF A GENETIC ALGORITHM

3.2.1. INDIVIDUAL

In nature, a set of organisms from the same species defines a population and each organism is described by a set of encoded instructions on its DNA. Each chromosome sets specific traits such as eye colour, and each trait can have multiple values: blue, brown, green, etc.

In a GA an individual does not represent an organism but a possible solution for the optimisation problem. The way it solves the problem is encoded in a convenient datatype where each characteristic of the solution is stored, similar to DNA in natural organisms. In general, an individual shares with the others the structure of its chromosome, but the values stored change from individual to individual (Figure 3.2).

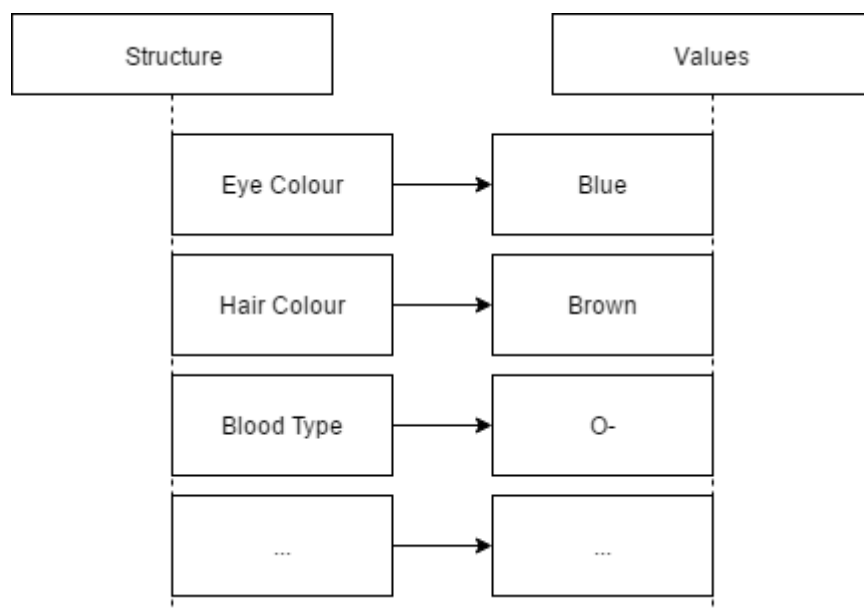


Fig. 3.2 – Chromosome structure and values

The original representation of a chromosome is a simple bit string vector – a series of 0 and 1 (Figure 3.3) – but even though it is the classical representation it is not always the most useful. For different problems, other datatypes can be better suited to store the characteristics of each individual, for example, a given problem might be better solved using individuals where the DNA is stored in a

multi-dimensional array and where the values stored do not necessarily use the base 2 (binary) numeric system.

[1 0 1 1 0 1 1]

Fig. 3.3 – Basic bit string chromosome

The datatype choice is problem dependant, the number of different variables that are changed during the optimisation and their relationship with each other, can exclude certain datatypes and make others more attractive for the developer. As an example (Figure 3.4), when optimising a structure's sections a vector can be used to represent the section in each bar, but if in addition to the section, the node coordinates (x,y,z) also need to be optimised, an array is probably the best way forward to represent the DNA of each individual solution.

<i>Number</i>	<i>x</i>	<i>y</i>	<i>z</i>	<i>Section</i>
1	1.1	0.0	10.0	<i>IPE100</i>
2	0.9	4.1	25.3	<i>IPE200</i>
3	0.3	1.5	7.3	<i>IPE200</i>
4	2.2	2.8	5.0	<i>L90x90x9</i>
5	1.0	3.0	3.0	<i>IPE140</i>

Fig. 3.4 – More complex chromosome datatype

3.2.2. INITIAL POPULATION

The initial population is the beginning of the GA. If there is previous knowledge of where the optimal solution is located in the search space, the initial population can be seeded near that position to save time in the search process. When there is no information that points the developer in a specific direction to where the ideal solution is located, a random set of individuals is generated to create the initial population (Figure 3.5).

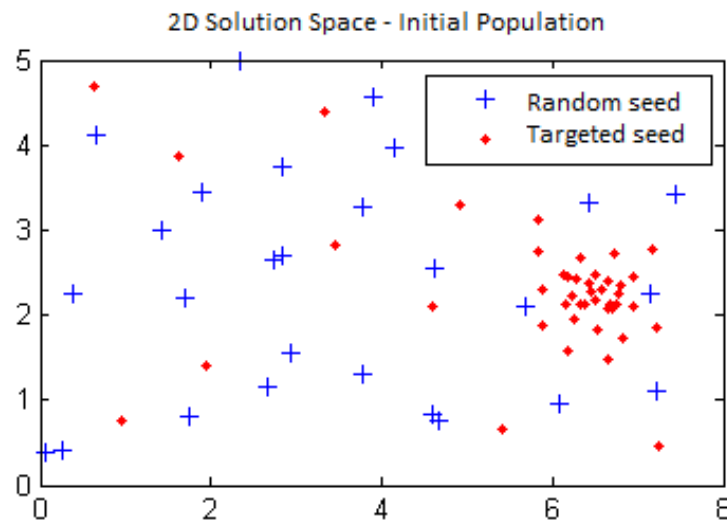


Fig. 3.5 – Random vs. Seeded population

The size of the initial population usually increases with the number of genes the GA operates with, in the search of the optimum solution. Using the example referred above, the GA that only optimises the bar sections will need a smaller population size than the second GA that optimises the section and

topology of the structure, in order to populate the solution space as well as the first GA in the previous example.

3.2.3. FITNESS FUNCTION

The Fitness function is a critical part of the GA and, it is responsible for the merit evaluation of each solution so that the best individuals have a larger probability to pass on their genes. To do that successfully, it needs to evaluate each individual and give penalties for bad solutions to decrease reproduction probability.

In simple problems, when there is a single and clear objective for the optimisation, the definition of the fitness function can be straightforward. For example:

-Arrange various shapes to be cut in a fabric of $W \times H$ (Width and Height) in order to reduce material waste. Fitness function: Evaluate the wasted area of each solution, apply penalties when the shapes overlap.

For more complex problems the first challenge arises, the most common is the optimisation of more than one parameter.

When more than one parameter is evaluated by the fitness function there needs to be a way to translate that multiparameter evaluation in a single classification that represents the quality of the solution. Such that operation can be quite difficult, for example:

When optimising a structure in addition to the material usage there might be specific nodes in the structure that cannot deflect more than a given value. In this case, several questions can be raised:

- How to weight the rigidity and mass parameters of the structure?
- When deflections are checked, is every analysed node equally important?
- How to add a penalty for a structure which has acceptable stiffness but poor material usage?
- Is the penalty so harsh that it can turn the algorithm into a random search?

The second challenge, which is very common in engineering problems, is the impossibility to evaluate directly each solution. Most GAs applied to engineering problems will need to evaluate each solution in FEM or CFD software and retrieve the results to evaluate the solution. For example, to optimise the drag coefficient of a car a call to an external CFD program needs to be made to get the drag coefficient of each solution. This requires external calls to complex software packages that can significantly delay the GA, making computational power and time an important constraint in such problems.

The fitness evaluation can be static or it can vary at runtime. As the generations increase, that change in evaluation is useful in very sensitive problems where the scale of penalties awarded initially to arrive at a near optimum are inadequate for the final adjustments needed to arrive at the final solution.

When it is impossible to define a fitness function, the adoption of interactive GAs might be an appropriate solution depending complexity of the problem. In interactive GAs, the fitness function is in fact the user that is asked to rate each individual. This is a useful approach when the algorithm is trying to optimise a problem classified with a subjective parameter such as aesthetics.

3.2.4. SELECTION FUNCTION

The selection function is used in conjunction with the fitness function to translate the Darwinian concept of survival of the fittest to the optimisation algorithm.

There are various ways to implement the selection function. The most commonly used are the roulette wheel selection and tournament selection, both are explained in the next subchapters. Even though there are clear differences between the various possible algorithms, the final goal is the same, i.e., to ensure that the individuals with better fitness classification are selected more often so that their genes are carried over to the next generation. When repeated over generations, it acts as a filter for the bad genes.

3.2.4.1. Roulette wheel

The roulette wheel algorithm (Figure 3.6) is the most commonly used method in traditional GAs. The key operations are:

- Sort the individuals according to their fitness value;
- Add the fitness values of each individual in the population and store the value:

$$Total = \sum fitness \quad (3.1)$$

- Assign to each individual a selection probability:

$$p = \frac{fitness}{Total} \quad (3.2)$$

- Randomly generate a number between 0.0 and 1.0;
- Sum the probability of each individual until the generated number is smaller than that sum;
- The last added individual is one of the parents of the next generation.

The implementation is straightforward and is effective in problems where convergence is easily attainable. When there is a need for more control on the selection function, the tournament selection is a better option.

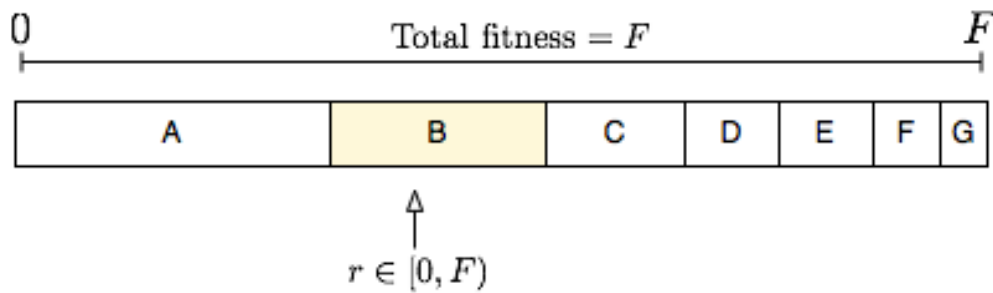


Fig. 3.6 – Roulette wheel: A is better than G therefore selected more often

3.2.4.2. Tournament selection

This method as the name implies is based on a tournament between individuals. A pool of randomly selected individuals from the population is created, and their fitnesses are compared. That tournament can return the best individual or a probability can be set for the selection of the best individuals, for example: the best individual is selected 70% for the time, the second 20% and the third 10% (Figure 3.7).

By having two adjustable parameters it is possible to fine-tune the selection algorithm to better suit the problem. An increase in the pool of individuals increases the selection pressure, decreasing the probability of the worse individuals being selected and by changing the probability of selection among the best individuals, early convergence is avoided.

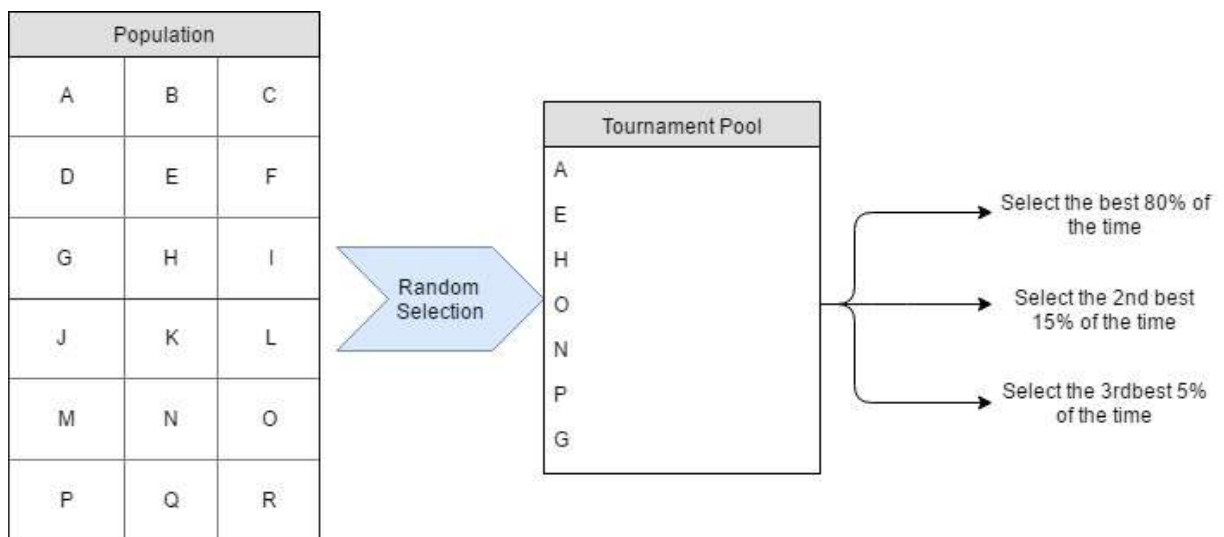


Fig. 3.7 – Tournament selection

3.2.5. GENETIC OPERATORS

Genetic operators receive the chromosomes from the parent individuals, and with manipulation of genes return a new individual inspired on its parents. In many cases the crossover operator is an acceptable way of solving a problem using GAs. However, it is common to add a mutation operator to ensure the diversity of the solutions. (Fogel, 2006).

3.2.5.1. Crossover

The crossover operator, receives the gene blocks from the parents and assembles them to create a new chromosome made of gene blocks from each parent that is assigned to the new individual.

There are several methods to create a new chromosome, the most common are the single and double point crossover that will be detailed next. The uniform crossover is another method that widens the search space of the GA. Due to this property, it is only suited for simple problems where the starting search space is not too large or when constraints such as computational capacity and time are not relevant.

The single and double point methods (Figure 3.8) are very similar, the chromosome of one parent is cut at a specific point and from that point onwards the chromosome is replaced by genes from the

other parent. The difference between single and double point is in the final point of mutation. Whilst in the single point crossover the genome uses genes from the second parent until the end of the genetic code, in the double point crossover, the replacement of genes is conducted until a generated end point.

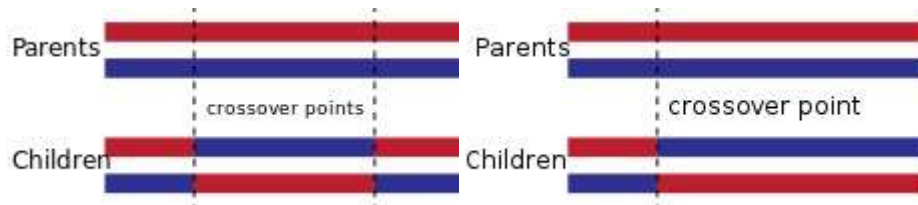


Fig. 3.8 – Double and single point crossover

3.2.5.2. Mutation

The mutation operator adds new information to the chromosome that is received from the crossover function.

This operator changes information randomly on some genes of the chromosome that is received from the crossover function. The objective is to preserve diversity in the solution space so that early convergence to a local optimum is prevented.

The operation is defined as a probability that each gene has of being changed to a random value. That probability needs to be low enough to ensure that the search does not degrade into a random search of the optimum.

The way the mutation is achieved depends on the datatype used to store the chromosome. For the classic bit string format, the mutation is achieved by a simple random change from 0 to 1 in some places of the string.

Mutation when the gene values are not in base 2 (binary) needs to be carried out with caution to avoid excessively drastic value changes that are particularly bad in the final phases of the GA when the fine-tuning of the solution is underway. To have that additional control, a method like the Gaussian mutation with the mean in the current value of the gene is desirable. This algorithm has been adopted in the present work and will be detailed in Chapter 4.

3.2.6. TERMINATION CRITERIA

There needs to be a way of informing the algorithm to stop the search. Such instruction is issued by the termination criteria. The developer has some leeway when defining this block of code but some circumstances might need to be considered. Termination examples are:

- Maximum number of generations;
- Found a sufficiently good solution;
- No recorded improvement over the last X generations;
- Computational time limit exceeded.

4

Program description

4.1. SUMMARY

In this chapter, the key elements of the application and adopted methodology in their development are detailed and the communication between the various components of the program is explained.

The program has two key elements, the Genetic Algorithm and an interface used to make calls to the FEM software used in the fitness evaluation, Autodesk Robot Structural Analysis.

The communication component was developed to make use of the API (Application Programming Interface) provided by Autodesk, this API allows a great level of control of most of Robot functions from inside any application developed in a language with support for COM interfaces (C++, C#, VB, etc.)

The option was made to use C# as the programming language as it offers a good balance between performance and development time. By not using C++ the manual memory allocation control was avoided and with it, likely memory leak problems were averted allowing more time to be focused on the development of the GA logic. Higher level languages such as Python and VB were avoided to improve the performance of the final program.

Rhinoceros and Grasshopper were used as auxiliary software in the initial stages of development to help define the algorithm responsible by the base DNA generation. The parametric design capabilities of grasshopper helped save time while defining the base tower structure geometry. This process will be detailed in the next subchapters.

4.2. PROGRAM FLOW

In this subchapter, focus is given to the communication between different components. Each component will be detailed just enough to make sense of the overall program flow, detailed description of each component is provided in the next subchapters.

The first interaction with the program is in the “Section Definition” tab (Figure 4.1 left) where the user is asked to add properties of the sections the GA will use in the search of the optimum. The sections are automatically added to the robot instance that is initialized with the program. With the sections defined, the user is then asked about geometric constraints (Figure 4.1 right) of the tower structure such as distance between ground supports, height of the power cables and number of arms.

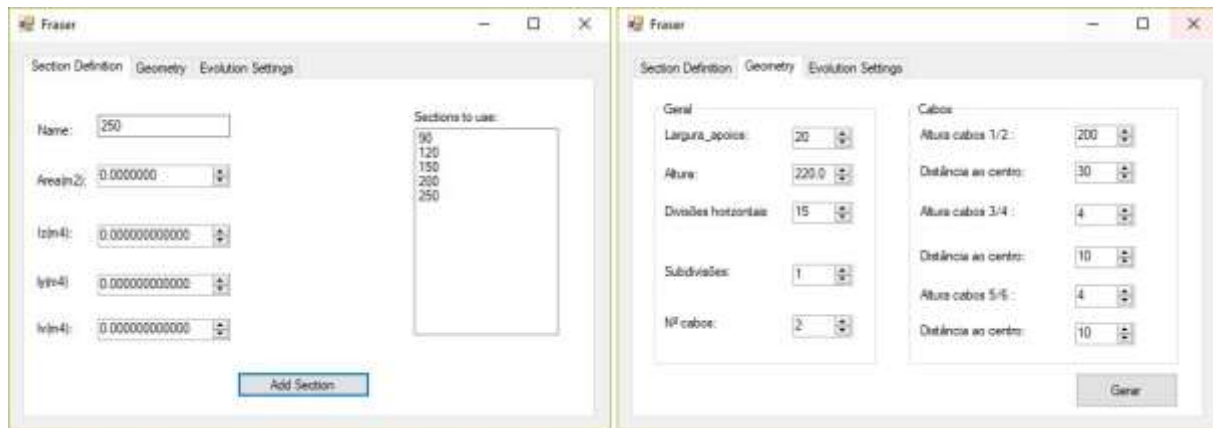


Fig. 4.1 – User interface

With the section and geometric information set, the base structure, as defined by the genetic code, can be generated. This is a structure with all the possible bars active. To achieve this, every node is connected to the other nodes immediately above and below in its own plane. That initial structure is therefore a highly redundant structural solution that is ready to be optimised by the genetic algorithm.

After the creation of the base structure, the user can define load cases using the Robot UI. Those load cases will be applied to every individual in the population, the self-weight is automatically adjusted for each solution. With the setup process complete, the initial population is created with a size previously set by the user.

For the initial population the x, y and z coordinates are randomly mutated and the section of each bar is randomly selected from all the sections defined plus an extra section that translates to a disabled bar in the Robot model.

A structural analysis is carried out for each individual. A list of bar forces is retrieved and used as input for the fitness function.

To determine the buckling lengths of the different bar types (Leg, Bracing, Horizontal bracing) according to the norms, the geometric information from the mutated base structure is processed by an auxiliary algorithm that runs through each set of bar types, checking which bars are connected at the end nodes and compiles that information in a calculation list that defines the internal analytical model (IAM) that will be used for the Eurocode checks. The list has, in each line, a new analytical bar and which real bars in the model make up that bar. The buckling length is also stored. When a bar has different buckling lengths in plane and out-of-plane two entries to the list are added.

With the calculation list assembled, the fitness function is executed. For each individual, the Class “Calc_operations” is responsible for running the EC3 checks and returning the utilization factor (U/f) of each bar. Three new lists are created: Bars with low U/f ; Bars with acceptable U/f ; Bars that fail the EC3 checks.

A repair operator is used in the chromosome to increase or reduce the section sizes for the under designed and over designed bars, respectively. When a bar fails the EC3 checks and is already the maximum section size possible, a penalty is added by the repair function to the overall weight of the structure. Over-designed bars do not need this penalty as they already add unneeded weight to the tower.

The fitness value (in metric tonnes) from the real structure is added to the fitness returned by the EC3 checks with weight penalties added. The final value is the final merit classification of each individual.

With the initial population evaluated, the selection function chooses parent elements to use as input to the “Breed” function. With the parents selected, the genetic operators are called. First, the crossover and then the mutation operator build the chromosome of the new individual and an automatic call is issued to update the FEM model and run the fitness calculation again. The new individual replaces the worst individual from the population pool and the Breed function is called again until the termination criteria is met. Figure 4.2 tries to explain graphically the interactions between key functions described above.

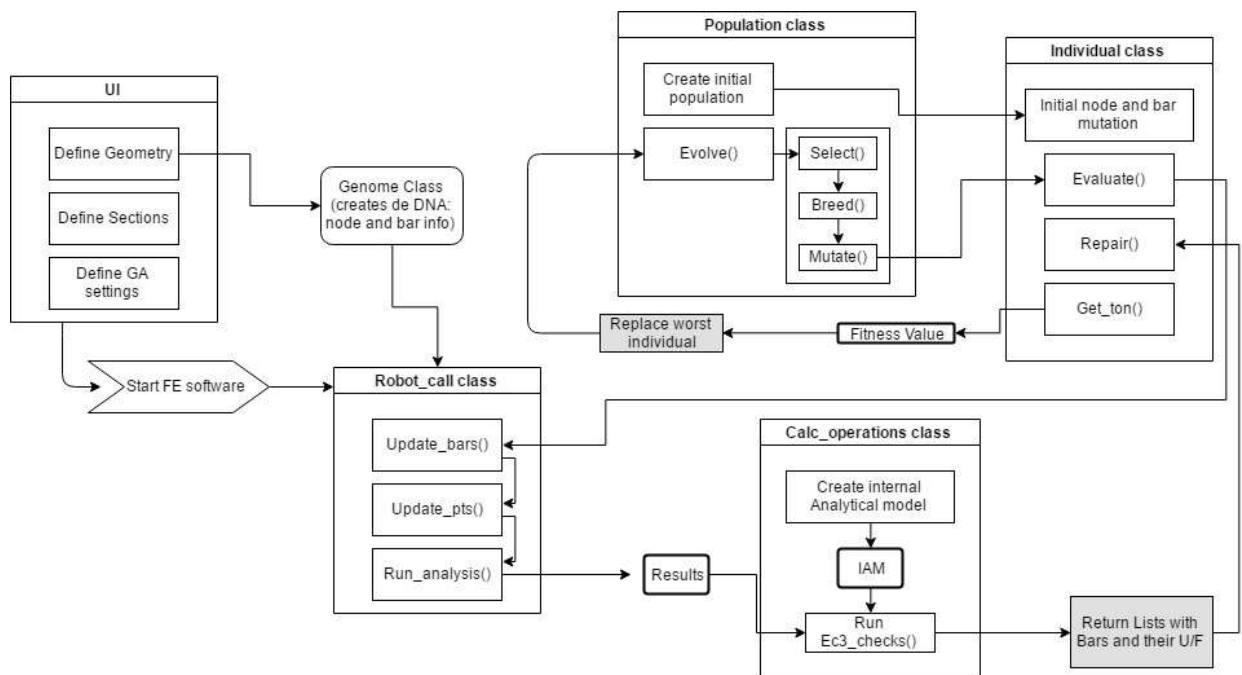


Fig. 4.2 – Interaction between key modules

An important aspect of the program that can be clearly seen on figure 2.4 is its modularity. By developing new components for the Genome and Calc_operations class, a completely different structure, that abides other design codes can be optimised by the algorithm. Also, by changing the Robot_call class, the program can use another FE software to analyse the structure.

4.3. BASE GENETIC CODE

The initial definition of a base structure is critical to the algorithm, an incorrect definition can lead the search in the wrong direction returning inadequate solutions. A base structure that is too restrictive stops the algorithm from exploring new search paths and a base structure with too much “freedom” could expand the search space beyond what is acceptable with today’s computing power.

To arrive at a base model that did not restrict too much the solution space, the base genetic code defined a structure with every node connected to its nearest nodes in the same plane. This resulted in a model with all the acceptable connections enabled. From there the genetic algorithm can decide which bars to delete and which node coordinates to change.

Rhinoceros and Grasshopper were used to develop the algorithm that generates the base structure. Grasshopper (Figure 4.3) is commonly known for its visual programming ability. However, to easily port the algorithm into the main application, the “C# scripting” component was used and the visual programming elements were reduced to input variables such as height and number of cables.

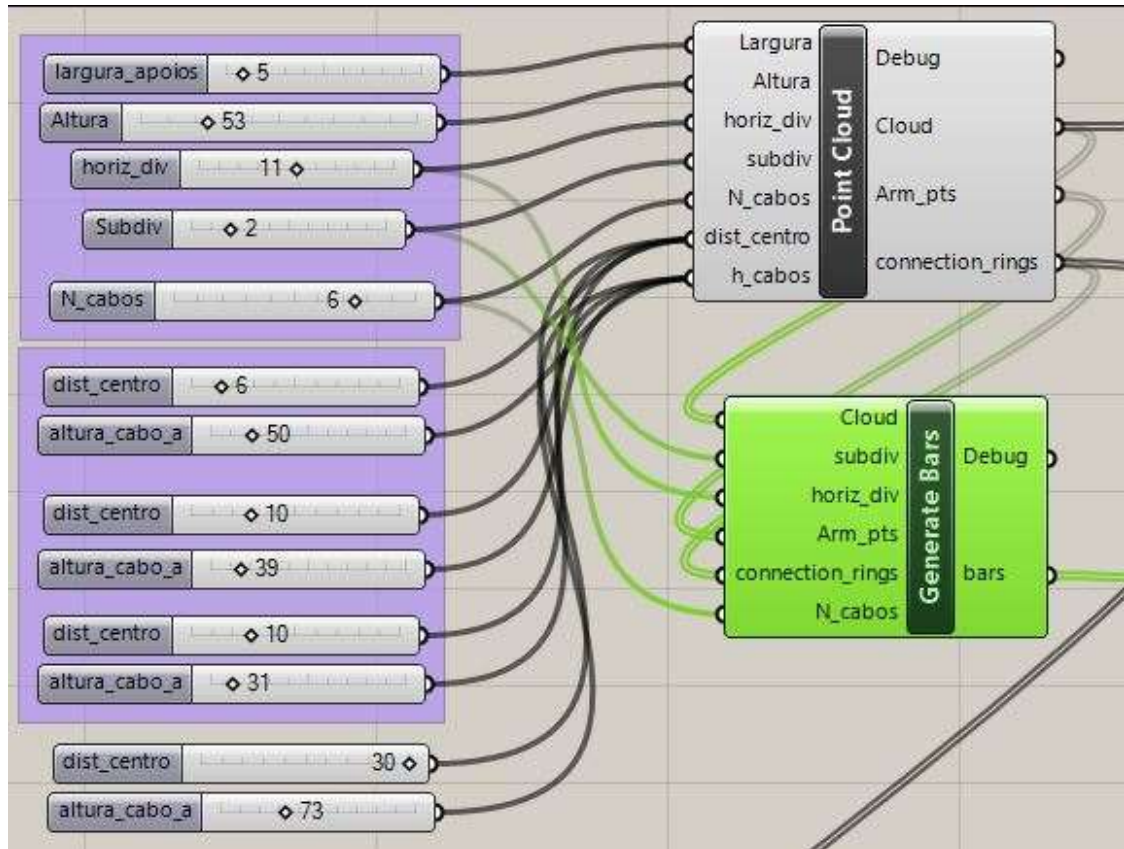


Fig. 4.3 Grasshopper

Using both programs helped streamlining the development of the geometry definition algorithm as any change promptly updated the model in Rhino (Figure 4.4) without the need to compile and run the code.

To port the final code into the main application the C# code was copied to the Genome class and the input variables were linked to the user interface elements.

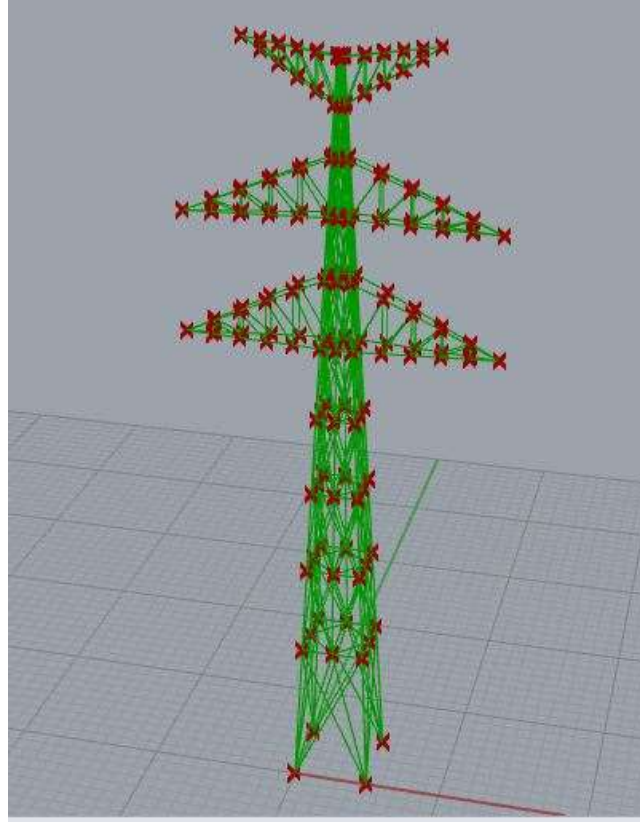


Fig. 4.4 – Grasshopper and Rhino model

With the code responsible for generating the geometry successfully ported, the node and bar elements needed to be placed in an adequate datatype that is easy to manipulate by the mutation and crossover functions. The datatype chosen is an array for nodes and bar elements each row with the following structure:

$$\text{Nodes: [Number | } x \text{ | } y \text{ | } z \text{ | mutation constant]} \quad (4.1)$$

$$\text{Bars: [Number | Point 1 | Point 2 | 1 or 0 | Section | id]} \quad (4.2)$$

The node array stores the node number, cartesian coordinates and a mutation constant. This constant gives more control over the mutation in nodes where the normal mutation scale is not incremental enough, this value scales the mutation for critical nodes. For example, in the arm nodes the mutation needs to be scaled down so that drastic topology changes do not occur from one generation to the other. Such behaviour could prevent the GA from reaching the optimum as each mutation could overshoot the best solution.

The bar array defines the bar number, the start and end nodes. The fourth column stores a 1 or 0 if the bar can be deactivated or not – leg and arm members cannot be deactivated – and the fifth column is an identifier, that allows the algorithm responsible for the buckling length calculation and EC3 checks to identify which type of bars it is analysing – legs have different calculation steps than bracing or arm bars.

An excerpt of the DNA definition algorithm is presented next.

Table 4.1 – DNA definition

```

public Genome(double Largura,int Altura, double horiz_div,double subdiv, int N_cabos, int[]
h_cabos,double[] dist_centro)
{
    (...)

    // init node and bar array
    pt_cloud = new double[5, (int)(17*N_cabos + 4 + (4 * subdiv) * (horiz_div - 1))];
    bars = new double[6, (int)((4 * horiz_div - 8) * (subdiv * subdiv) + (12 * horiz_div - 12)
* subdiv - 8 * horiz_div + 36 * N_cabos + 20)];

    // function call for the geometry generation methods
    pt_add_tower(ref pt_cloud, Largura, Altura,horiz_div,subdiv,ref pt_cnt);
    pt_add_arms(ref pt_cloud, Largura, Altura, horiz_div, subdiv, N_cabos, h_cabos,
dist_centro, ref pt_cnt, ref connection_rings);
    bar_cnt = connect_bars(ref bars,(int)subdiv,(int)horiz_div);
    add_arm_bars(ref bars, ref bar_cnt, connection_rings, (int)subdiv,
N_cabos,(int)horiz_div);

    // store number of tower bars (subtract the arm bars)
    towerBar_cnt = bar_cnt - 36 * N_cabos;
}

private void pt_add_tower(ref double[,] pt, double Largura, int Altura, double horiz_div, double
subdiv,ref int _pt_cnt)
{
    (...)

    //#####//
    //main pt cloud loop//
    //#####//

    for (h = 1; h <= horiz_div - 1; h++)
    {
        double scale_factor = (1 - (h / horiz_div));
        double step = h * tilt;

        if (!reverse)
        { // x++ y++

            for (x = 0; x <= subdiv; x++)
            {

                addPt(ref pt, pt_num, step + x * (Largura / subdiv) * scale_factor, step, h *
ring_z_step,(double)Altura);
                pt_num++;

                if (x == subdiv)
                {

                    for (y = 1; y <= subdiv; y++)
                    {
                        addPt(ref pt, pt_num, Largura - step, step + y * (Largura / subdiv) *
scale_factor, h * ring_z_step, (double)Altura);
                        pt_num++;

                        if (x == subdiv && y == subdiv) { reverse = true; }

                    }

                }

            }

        }

    }

    (...)

private int connect_bars(ref double[,] bars,int subdiv, int horiz_div)
{
    int bar_num = 0;
    //Support pts
    for (int i = 0; i <= 4; i++)
    {

        if (i == 0)
        {

```

```

        for (int j = 4; j <= 4 + subdiv; j++)
        {
            if (j == 4)
            {
                addBar(ref bars, bar_num, 0, j, 0, 0,0);
                bar_num++;
            }
            else {
                addBar(ref bars, bar_num, 0, j,1,0,1); // id =1 bracing
                bar_num++;
            }
        }
        for (int j = 8 + 4 * (subdiv - 1) - 1; j >= 8 + 4 * (subdiv - 1) - subdiv; j--)
        {
            addBar(ref bars, bar_num, 0, j, 1, 0,1); // id =1 bracing
            bar_num++;
        }
    }
    (...)

```

4.4. INITIAL POPULATION

The initial population is created when the user defines the initial geometry. As mentioned before, the size of the population is initially defined by the user.

The size of the population needs to be set as a function of the number of bars and sections available. A simple structure with 200 bars and only two or three types of cross-section sizes does not require the same number of initial random individuals to adequately populate the solution space as a highly complex tower with 1000s of bars and 10s of possible cross-sections sizes.

The initial population has several random individuals. Those individuals all start as the same base structure described in Section 4.3. To add randomness to their properties, an initial mutation function is applied to the nodal coordinates and bar sections. The mutation is introduced with the random seeder that is available in the .Net library, and is implemented in the next code snippet:

Table 4.2 – Initial mutation implementation

```

public Individual(Genome _baseDNA, ref Random rndm)
{
    this.fitness = 0.0;

    _DNA = new Genome(); // create new chromosome

    //if the following is not done the same matrix is always changing (by ref vs by value):
    _DNA.pt_cloud = (double[,])_baseDNA.pt_cloud.Clone(); // copy by value the pt_cloud[]
    _DNA.bars = (double[,])_baseDNA.bars.Clone(); //copy by value the bars[]

    for (int i = 4; i < Genome.pt_cnt; i++) // start at 4 to fix supports
    {
        //mutate initial pt coord.
        _DNA.pt_cloud[1, i] += rndm.Next(-1, 1) * rndm.NextDouble() * _DNA.pt_cloud[4, i]; //X
        _DNA.pt_cloud[2, i] += rndm.Next(-1, 1) * rndm.NextDouble() * _DNA.pt_cloud[4, i]; //Y
        _DNA.pt_cloud[3, i] += rndm.Next(-1, 1) * rndm.NextDouble() * _DNA.pt_cloud[4, i]; //Z
    }
    //define init sections
    for (int i = 0; i < Genome.bar_cnt; i++)
    {
        if (this._DNA.bars[4,i] == -1)
        {
            this._DNA.bars[4, i] = 0;
        }
    }
}

```

```

    }else
    {
        this._DNA.bars[4, i] = Population.rand.Next(0, Sections.count-1); // rand. section
    } (...)

```

With each individual randomly mutated, the Evaluate function, which is described in the next subchapter, can run on each element of the initial population. The following code snippet calls the Evaluate function for the initial population:

Table 4.3 – Evaluate function call

```

for (int i=0; i<max_generations; i++)
{
    Generation.Text = i.ToString();

    if (i == 0)
    {
        for (int a = 0; a < Population.Pop_size; a++)
        {
            CurrentPop.ind[a].Evaluate();//Evaluate call
            c++;
            series.Points.AddXY(c, CurrentPop.ind[a].fitness); //Plot graph
        }
    }
}
(...)

```

4.5. EVALUATE FUNCTION

This function, is responsible for the main part of the intensive calculations needed to calculate the fitness value of each solution and it encloses a series of methods.

It starts by updating the model in Robot, adding the supports and load cases, and then runs the analysis and stores the results.

With the results stored the creation of the internal analytical model (IAM) is started. It runs through every bar type, checks end connections and translates that information into calculation lists that are ready to be used by the EC3_check function. The next snippet is from one of the methods that scans through the structure and creates the IAM.

Table 4.4 – IAM initial scan (leg scan excerpt)

```

List<Int32> temp = new List<Int32>();

for (int i = 0; i < Genome.horizd-1; i++)
{
    if (i == 0) // init bar
    {
        temp.Add(i + 1);
        //if horiz bar/bracing/ change section
        if (v_braced(1, new List<Int32>() { 1, 2 }))) {

            Leg_ops.Add(new Calc_operations(1, temp, (int)_DNA.bars[4, 0],
(int)_DNA.bars[5, 0])); // add to IAM list for leg calcs
            temp = new List<Int32>();

        } else if (_DNA.bars[4, 0] != _DNA.bars[4, (8 * (int)Genome.subd) + 4])
        {
            Leg_ops.Add(new Calc_operations(1, temp, (int)_DNA.bars[4, 0],
(int)_DNA.bars[5, 0]));
            temp = new List<Int32>(); //Reset
        } else{// go to next bar}
    } else {

        int bar_ind = 8 * (int)Genome.subd + 5 + (i - 1) * (4 * (int)Genome.subd *

```

```

(int)Genome.subd + 8 * (int)Genome.subd - 8);
    if (v_braced(bar_ind, new List<Int32>() { 1, 2 }))
    {
        if (temp != null) // if not null start bar = temp[0]
        {
            temp.Add(bar_ind); // add bar
            Leg_ops.Add(new Calc_operations(temp[0], temp, (int)_DNA.bars[4, bar_ind-
1], (int)_DNA.bars[5, bar_ind-1]));
            temp = new List<Int32>();
        } else
        (...)
    }

```

There are other methods to create the IAM for bracing and horizontal bar calculations, all following the same general workflow: The algorithm runs through adjacent bars of the same type and, at each intersection with bar elements, it checks if that type of intersection provides bracing, if it does a list entry is added with a new analytical bar number and the real elements that makeup that bar, from that information the buckling length is added. The function also checks for different in-plane and out-of-plane buckling lengths.

The final list is used as input for the EC3 checks and has the following format:

$$[initial\ bar|List\{unbraced\ real\ bars\}] \text{ buckling length} \quad (4.3)$$

To better understand how the algorithm interprets the structure and builds the IAM, Figure 4.5 provides two structural solutions, one (left) where the horizontal bracing elements were needed and one (right) where a similar structure in which those horizontal bars were deemed unnecessary.

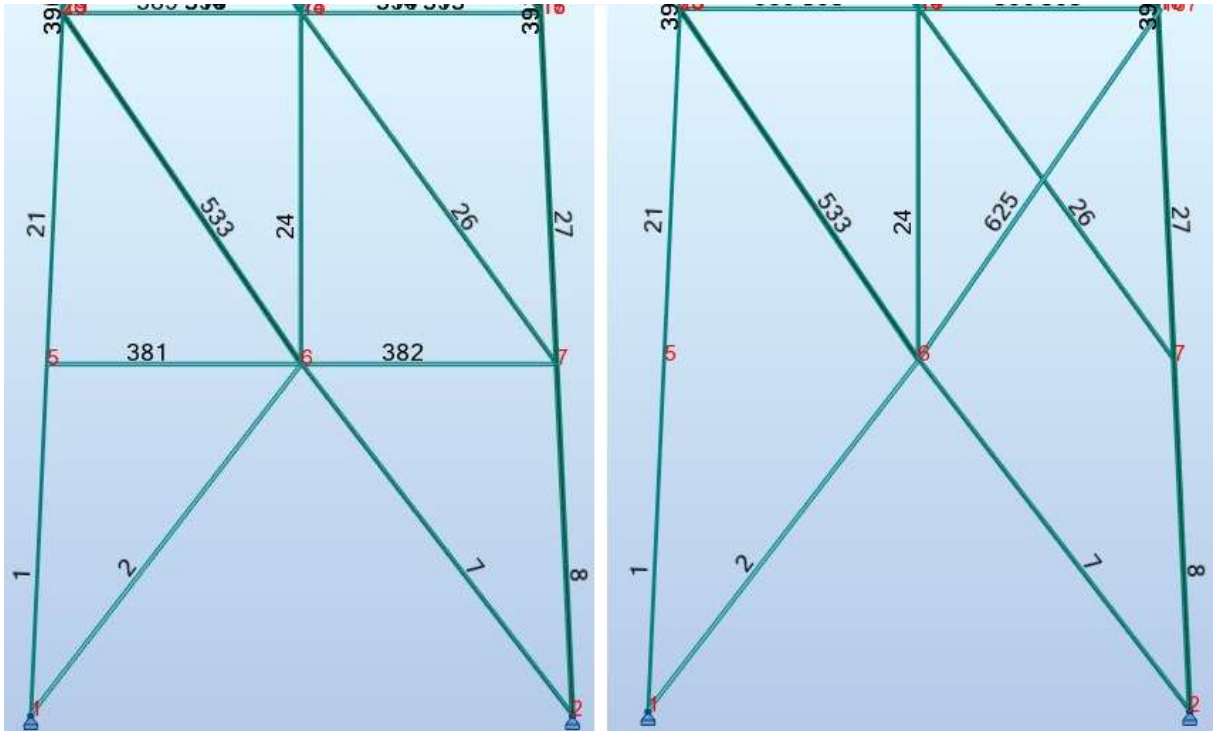


Fig. 4.5 – IAM: Different bracing conditions

For the structure on the left the following four entries would be added to the IAM for the leg members:

Left leg: [1|{1}| *Length of robot bar 21*] and [21|{21}| *Length of robot bar 21*]

Right leg: [8|{8}| *Length of robot bar 8*] and [27|{27}| *Length of robot bar 27*]

For the structure on the right, the left leg would become only one entry in the IAM while the right leg is still braced by one diagonal bar and keeps the same two entries listed above:

Left leg: [1|{1,21}| *Length of robot bar 1 + 21*]

Right leg: [8|{8}| *Length of robot bar 8*] and [27|{27}| *Length of robot bar 27*]

Regarding the Eurocode 3 checks, which have already been described in Chapter 2, the following code is an excerpt of the verification translated into code for the leg elements. Similar code is present in the EC3_checks function for other bar types.

Table 4.5 – EC3 verification for leg members

```

/// Leg calc excerpt    ///
/// BS EN 1993-1-1:2005  ///
/// BS EN 1993-3-1:2006  ///
///#####///

for (int i = 0; i < calc_ops.Count; i++)
{
    double Nsd = 0;
    double L = 0;

    // get max N of every bar //
    for (int b = 0; b < calc_ops[i].next_bars.Count; b++)
    {
        if (Math.Abs(results[2, calc_ops[i].next_bars[b] - 1]) > Nsd) { Nsd =
results[2, calc_ops[i].next_bars[b] - 1]; }
    }
    // get total L for buckling //
    for (int b = 0; b < calc_ops[i].next_bars.Count; b++)
    {
        L += results[1, calc_ops[i].next_bars[b] - 1];
    }

    if (Nsd > 0) //tension
    {
        double u_f = 0;
        double Nu_rd = Sections.Area[calc_ops[i].section_id] * 275000; //Aeff * fy
        u_f = Nsd / Nu_rd;

        for (int y = 0; y < calc_ops[i].next_bars.Count; y++)
        {
            repair_instructions.Add(new double[] { calc_ops[i].next_bars[y],
Math.Abs(u_f) }); // add to repair list (each individual bar)
        }
    }
    else { //compression

        //resistance check
        double u_f = 0;
        double Nc_rd = Sections.Area[calc_ops[i].section_id] * 275000;
        u_f = Nsd / Nc_rd;

        //buckling check
        double lambda = L / Sections.ivv[calc_ops[i].section_id]; // L/ivv
        double _lambda = lambda / (93.9 * Math.Sqrt(235 / 275));
        double k = 0.8 + (_lambda / 10);
        if (k > 1) { k = 1; }
        if (k < 0.9) { k = 0.9; }
        double _lambda_eff = k * _lambda;
        double fi = 0.5 * (1 + 0.34 * (_lambda - 0.2) + _lambda * _lambda);
    }
}

```

```

double xi = 1 / (fi + Math.Sqrt(fi * fi - _lambda * _lambda));
double Nb_rd = xi * Sections.Area[calc_ops[i].section_id] * 275000;
double b_uf = Nsd / Nb_rd;

if (u_f < b_uf) { u_f = b_uf; }

for (int y = 0; y < calc_ops[i].next_bars.Count; y++)
{
    repair_instructions.Add(new double[] { calc_ops[i].next_bars[y],
Math.Abs(u_f) }); // add to repair list (each individual bar)
}
}
}

```

As shown by the code, each evaluated bar is added to a list that stores the utilization factor for that specific bar element. This list created by the code above, is used by the next function called by the Evaluate routine. This function stores bars in three lists based on the U/f – low U/f, acceptable U/f and oversized bars. This is done by the default sorting algorithm implemented for array objects in the .Net library.

4.6. REPAIR FUNCTION

The three lists containing bars with different utilization factors are sent to the repair function that runs before the final fitness evaluation.

A chromosome repair function is useful in constrained optimization problems to correct illegal gene values, to help move the solution towards a feasible region.

In this case, a gene value is deemed illegal when a bar section is insufficient to achieve a utilization factor below 1.0. When that occurs, the bar cross-section is increased if it is not already the largest section available. When a bar element with the largest section does not have a U/f of less than 1.0, a penalty value is added to the solution in the form of increased weight.

Several expressions were tested – considering the number of illegal genes, how much over 1.0 the U/f was and the state of convergence of the algorithm – but, in the end, a much simpler version of the penalty expression was the most successful. During testing it was concluded that if there are no abrupt size changes in the sections provided to the GA, a 0.25 tonne increment for every bar with maximum section and U/f over 1.0 would be a good penalty value for large structures (over 50 meters) and an increment of 0.15 ton worked well for smaller structures (50 or less).

To help steer the solution without constraining it to a local optimum, an additional method was also added for bars with excessively low U/f. As those bars are not illegal, a randomly selected subset of bars with low U/f have their sections reduced by that method.

When repair functions are used it is important to use them only to steer the search in the right direction. Not every gene can be changed by the repair function as that would guide the solution into a local optimum – defeating the purpose of a GA by transforming into an hill climbing optimization – rather that point towards the general direction of the solution.

The following is an excerpt of the repair function.

Table 4.6 – Repair function excerpt

```

private void Repair(ref List<double[]> ovr_dsgn, ref List<double[]> udr_dsgn, ref List<double[]>
_dsbl)
{
    int Section_count = Sections.count;

    ///Over Designed

    int bars_to_correct;
    bool need_bigger_sect = false;

    if (max_bars_to_reduce <= ovr_dsgn.Count) {
        bars_to_correct = Population.rand.Next(2, ovr_dsgn.Count/10);
    }else
    {
        bars_to_correct = Population.rand.Next(0, ovr_dsgn.Count);
    }

    for (int i = 1; i < bars_to_correct; i++)
    {
        double[] tmp = ovr_dsgn[i - 1];
        if (this._DNA.bars[4, (int)tmp[0] - 1] > 0) // if not the largest section available
        {
            this._DNA.bars[4, (int)tmp[0] - 1]--;
            Console.WriteLine("Decrease section" + (int)tmp[0]);
        }
        else {}
    }

    ///Under Designed
    for (int i = 0; i < udr_dsgn.Count; i++)
    {
        double[] temp = udr_dsgn[i];
        if (this._DNA.bars[4, (int)temp[0] - 1] != Section_count - 1){
            this._DNA.bars[4, (int)temp[0] - 1]++;
            Console.WriteLine("Increased section" + temp[0]);
        }
        else
        {
            this.fitness += 0.15;

            need_bigger_sect = true;
            Console.WriteLine("-----");
            Console.WriteLine("!!!NEEDS BIGGER SECTIONS!!!");
            Console.WriteLine("-----");
        }
    }
}

```

After this routine, the penalty value is added to the real weight of the structure to create the fitness value of the individual. The population is then sorted from worst to best individual.

4.7. SELECTION FUNCTION

The tournament selection was used for the selection function after tests with the roulette wheel selection revealed the lack of adaptation the algorithm provided. Being this a minimization problem, the implementation of the tournament selection was also straightforward.

As described in Chapter 3, the tournament selection algorithm works by populating a list of randomly selected individuals and selecting the best with a predefined probability.

In the present implementation, the probability of selecting the best individual from the list is set to 100%. This decision was made after confirming that changing the selection pressure (size of the list) provided sufficient control over early convergence into a local optimum.

The following code shows the implementation of the tournament selection described.

Table 4.7 – Tournament selection

```
public static Individual Tournament_selection(Individual[] pop)
{
    int selection_pressure = (int)Pop_size/6; // adjust selection pressure
    int[] tournament = new int[selection_pressure];

    for (int i = 0; i < selection_pressure; i++)
    {
        tournament[i] = Population.rand.Next(0, Pop_size - 1);
    }

    Array.Sort(tournament);

    return pop[tournament.Last<int>()];
}
```

4.8. GENETIC OPERATORS

4.8.1. CROSSOVER

The single point crossover was tested in a previous version of the code but it did not help convergence, particularly in complex structures in which the search degenerated into a random search. To prevent this from happening the double point crossover was used to reduce the number of genes that changed between each generation, making the evolution more incremental.

The next code snippet implements the crossover function applied to the bar array. The implementation is similar for the node array but the changed values are the x,y,and z coordinates.

Table 4.8 – Crossover function applied to the bar array

```
///BARS
///Crossover

CrossOver_pt = Population.rand.Next(0, Genome.towerBar_cnt);
int end_crossover_pt = Population.rand.Next(CrossOver_pt, Genome.towerBar_cnt);

for (int i = CrossOver_pt; i < end_crossover_pt; i++)
{
    ///X is the new individual that initially takes all the genes from the best individual
    ///Parts of the chromosome are changed for the other selected individual (b) with
    ///the following code:
    x._DNA.bars[4, i] = b._DNA.bars[4, i]; ///Change section
}
```

4.8.2. GAUSSIAN MUTATION

From the problems that were observed while trying to implement a simple single point crossover, it became clear that this structural optimisation problem is very sensitive to drastic changes in the genome. That was valuable information for the implementation of the mutation function.

As the values for each gene are stored as decimal values – instead of the classical binary representation – a mutation applied with the same random seeder that is used to create the initial population would prove to be a too drastic change to allow the correct search to be carried by the GA. This would be especially critical in the bar section mutation as an uncontrolled mutation could change a bar from having the largest section to a deactivated bar in the next generation.

To prevent this, the node mutation used a random seeder scaled down so that the maximum delta from the original x, y and z coordinates was lower than 10 cm. For the mutation of bar sections a Gaussian mutation operator was used.

The Gaussian mutation is commonly used to prevent drastic changes in the genome. It works by selecting the next gene value from a Gaussian distribution centred on the current value of the gene. The sigma value of the Gaussian distribution was found to be a parameter that needs to be adjusted based on the problem definition. Specifically, the number of sections that the algorithm can try on the structure, after defining the gaussian distribution, needs to be tapered at both ends to allow only possible values for the sections sizes.

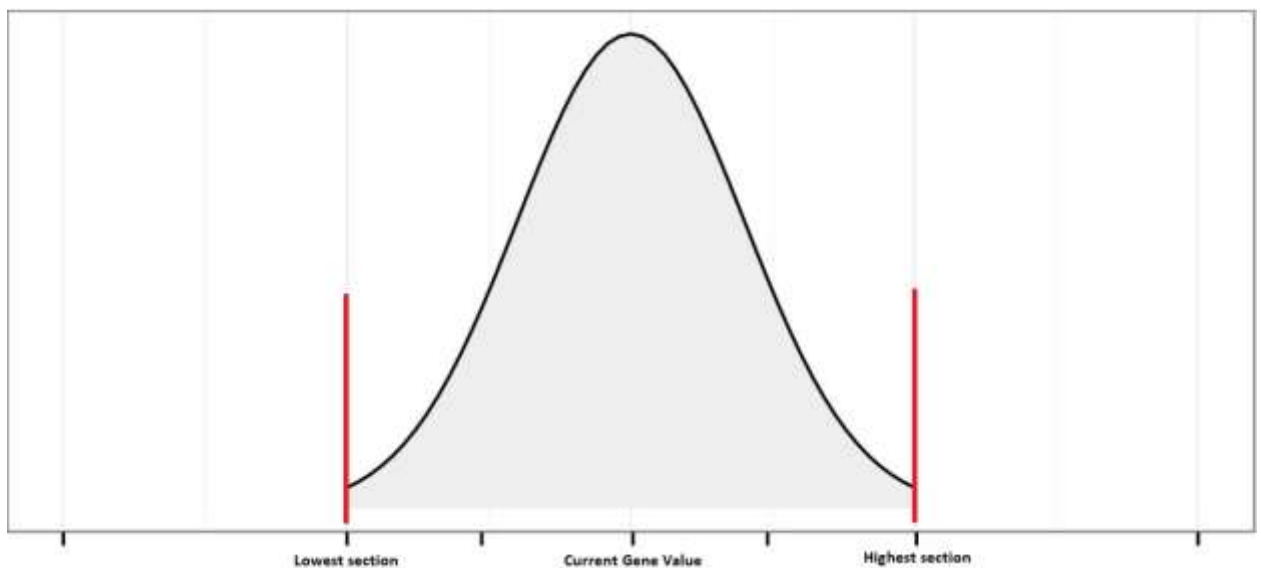


Fig. 4.6 – Tapered normal distribution

To prevent the need of additional libraries to plot a normal distribution, an implementation from NashCoding – a website dedicated to artificial intelligence – was used:

Table 4.9 – Gaussian mutation algorithm

```
private static double gaussianMutation(double mean, double stddev)
{
    double x1 = rand.NextDouble();
    double x2 = rand.NextDouble();

    if (x1 == 0)
        x1 = 1;
    if (x2 == 0)
        x2 = 1;

    double y1 = Math.Sqrt(-2.0 * Math.Log(x1)) * Math.Cos(2.0 * Math.PI * x2);

    return y1 * stddev + mean;
}
```

To limit the min and max values to the possible section values, from -1 to number of sections (-1 being a disabled bar) an extra function is called to correct values that fall outside that range:

Table 4.10 – Clamp function

```
private static double clamp(double val, double min, double max)
{
    if (val >= max) {return max;}

    if (val <= min) {return min;}

    return val;
}
```

Finally, the mutation function that calls both auxiliary methods listed above is presented in the next code snippet:

Table 4.11 – Mutation function

```
if (_rnd < sec_mutation_prob)
{
    if (x._DNA.bars[3, i] == 1)
    {
        double sigma = Sections.count / 2;
        double gene_val = gaussianMutation(x._DNA.bars[4, i], sigma);
        gene_val = clamp(gene_val, -1, Sections.count - 1);
        x._DNA.bars[4, i] = (int)gene_val;
        cnt++;
    } else
    {
        double sigma = (Sections.count-1) / 2;
        double gene_val = gaussianMutation(x._DNA.bars[4, i], sigma);
        gene_val = clamp(gene_val, 0, Sections.count - 1);

        x._DNA.bars[4, i] = (int)gene_val;
        cnt++;
    }
    if(x._DNA.bars[5,i] == 5) { x._DNA.bars[4, i] = 0; }
}
```


5

Case Study

In this chapter, the developed application will be used to design a lattice tower structure that withstands the same loads and has the same geometric constraints as a case study tower provided by Metalgalva.

In the next subchapters, the model will be presented, along with the steps taken to prepare the optimization process.

Finally, the results will be analysed and conclusions about real world implications will be drawn along with future development paths that could be taken to improve the application.

5.1. BASE MODEL

The case study tower is 38 meters high, and carries 7 cables, 3 in each side and one at the tip of the tower (Figure 5.1). The distance between leg members at the base is 5 meters in both directions and, each arm is 2.25 meters long. The steel used is of type S275. The model provided by Metalgalva has a total weight of 7.616 tonnes, distributed according to Table 5.1

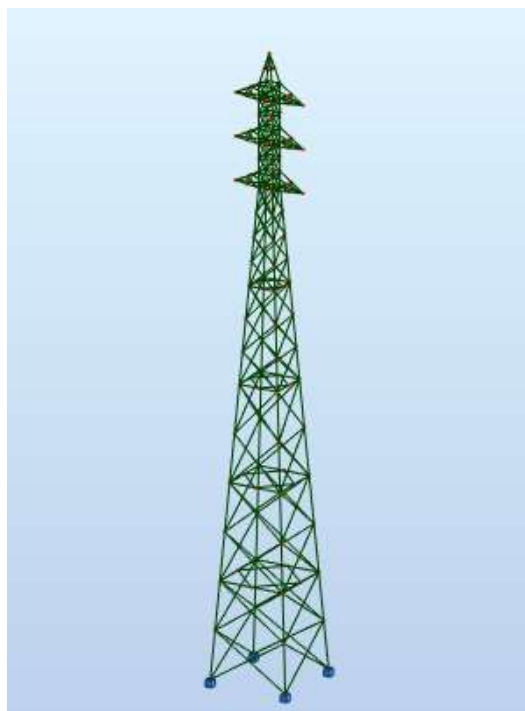


Fig. 5.1 – Base Model

Table 5.1 – Base model weight distribution

Sections	Number	Total weight (kg)
L 50x50x5	262	1873
L 60x60x6	44	699
L 70x70x7	4	123
L 100x100x10	4	379
L 140x140x13	4	660
L 160x160x15	4	870
L 180x180x6	4	1046
Ls 180x180x15	8	1967
		7616

Regarding the loads, the base model provided by Metalogalva contained hundreds of load cases. To improve the run time of the optimisation routine, the critical load cases were identified with the help of Metalogalva's technical department. This interaction with the engineering team, allowed the application to run with only 4 critical load cases added to the self-weight of the structure. Given the way the DNA of the structure is generated, this reduction of load cases adds the need of a final inspection of the output model to ensure symmetry. This is a point of further improvement of the program that will be detailed at the end of the chapter.

Figure 5.2 identifies the arm nodes and Table 5.1 shows the forces applied to each of the four critical load cases (LC1 to LC4).

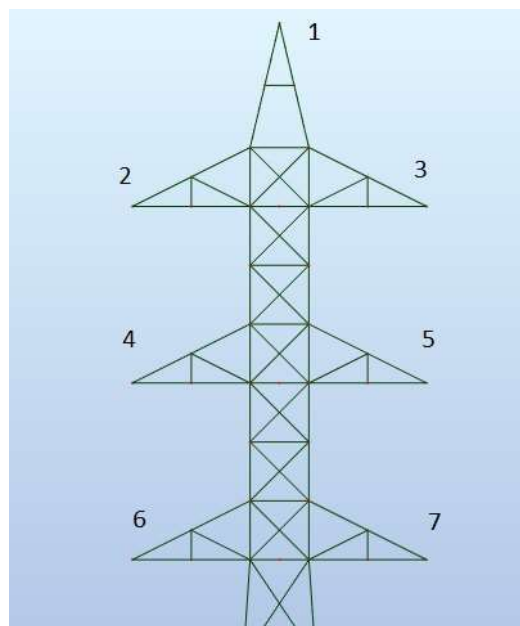


Fig. 5.2 – Arm nodes

Table 5.2 – Load cases

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7
LC1	FX=13.24	FX=24.02	FX=24.02	FX=24.02	FX=24.02	FX=24.02	FX=24.02
(kN)	FY=1.23	FY=0.49	FY=0.49	FY=0.49	FY=0.49	FY=0.49	FY=0.49
	FZ=-3.43	FZ=7.84	FZ=7.84	FZ=7.84	FZ=7.84	FZ=7.84	FZ=7.84
LC2	FX=0.0	FX=0.0	FX=0.0	FX=0.0	FX=0.0	FX=0.0	FX=0.0
(kN)	FY=12.50	FY=29.41	FY=22.06	FY=22.06	FY=22.06	FY=22.06	FY=22.06
	FZ=-2.45	FZ=-7.84	FZ=-4.90	FZ=-4.90	FZ=-4.90	FZ=-4.90	FZ=-4.90
LC3	FX=1.72	FX=2.45	FX=2.45	FX=2.45	FX=2.45	FX=2.45	FX=2.45
(kN)	FY=12.50	FY=22.06	FY=22.06	FY=22.06	FY=22.06	FY=22.06	FY=22.06
	FZ=-2.45	FZ=-4.90	FZ=-4.90	FZ=-4.90	FZ=-4.90	FZ=-4.90	FZ=-4.90
LC4	FX=0.0	FX=0.0	FX=0.0	FX=0.0	FX=0.0	FX=0.0	FX=0.0
(kN)	FY=12.50	FY=22.06	FY=29.41	FY=22.06	FY=22.06	FY=22.06	FY=22.06
	FZ=-2.45	FZ=-4.90	FZ=-7.84	FZ=-4.90	FZ=-4.90	FZ=-4.90	FZ=-4.90

5.2. PROGRAM SETUP

To configure the application to design an optimised structure that withstands the loads listed above and with similar geometrical characteristics, a few steps are needed.

As it is a small tower (38m), the penalty function, which was discussed on Chapter 4, is changed to 0.15 tonnes. A list of sections needs to be provided to the genetic algorithm before starting the optimisation. The sections considered were the following steel equal angles:

- L50x50x5
- L60x60x6
- L70x70x7
- L100x100x10
- L150x150x15
- L180x180x16

The list has fewer sections than the base model. This decision was based on a previous analysis in which it was concluded that there was no need, based on bar utilization factors and the expected optimization ratio, for two intermediate bars between the L 100 and L 180 sections. A single L 150 section was used instead of the L 140 and L160.

This reduction of available sections also allows the program to run faster as the search space is reduced. However, there is a risk that the output structure still has penalties applied, in other terms, some bars need to have larger sections that are not available. To address this issue, a log file containing the list of bars that need to be strengthened (either by secondary bracing or larger sections) is produced for the final structure.

The load cases described in Section 5.1 were added using the Robot UI and for each load case the self-weight of each bar was added. This additional load is automatically updated by Robot in each iteration.

5.2.1. TEST RUNS AND CALIBRATION

Given the nature of the program, it requires several parameters to be adjusted based on the structure characteristics. This calibration is done to ensure not only a thorough search of the solution space but also a reasonable computational time.

The first parameters to adjust were the threshold values of the U/f that defined over-designed and under-designed bars. This is highly dependent on the variety of sections provided to the GA and how incremental their ultimate resistances are.

The U/f threshold for bars that can be randomly reduced (Section 4.6), needs to be small enough so that a possible reduction or deactivation of a bar does not cause the failure of other bars in its proximities due to load redistribution. Moreover, this U/f value needs to be high enough to remove unnecessary bars from the structure. This is structure dependant as larger structures are likely to have more elements to resist load increases of this type.

To define this value, a test run in debug mode was carried out. Running in debug mode allowed for the U/f threshold to be changed at runtime. The reduced or deleted bars were monitored as well as the remaining bars for changes in U/f . After a few iterations, the lower threshold value of 0.1 was adopted for the majority of the search. In the final iterations, when all the bars are likely to exceed that U/f threshold, the value needs to be increased. In this case, it was increased to 0.3 for the final generations of the GA.

It is important to note that this lower U/f value is used to steer the solution in the right direction and not to limit the solution (Section 4.6). There can be active bars with U/f lower than 0.1 and inactive bars with U/f higher than 0.3 because of the mutation operator.

The higher U/f threshold that splits into two lists over and under-designed bars, was found in the same iterative way to ensure that for bars that failed the EC3 check, the next available cross-section has a high probability – to increase convergence speed – to pass EC3 checks. The value adopted was 0.7 for the entire optimisation run – bars with U/f larger than 0.7 are therefore not reduced.

The remaining adjustable elements regarding the GA algorithm itself are: population size, mutation probability and mutation pressure.

These three parameters are interdependent. For that reason, the calibration requires several test runs to find the ideal combination for the structure to optimize. There are a few rules of thumb useful for the calibration. For example, a low population size can be balanced by a high mutation probability (provided the algorithm still converges); a low selection pressure can be counteracted by a high population size; These rules of thumb have both the same objective, i.e., to maintain search space diversity.

For this case, given that during testing with small structures (200 bars and 3 sections) a population size of 25 provided good results, the population size for the case study was increased linearly – resulting in a population size of 225. From this point, the mutation probability and selection pressure were both adjusted.

With the default mutation probability of 15% (very high for traditional GAs), the search degraded into a random search. The second iteration, with a mutation probability of 10% returned some form of convergence. However, from the several peaks and troughs visible in Figure 5.3, it was apparent that the solution space was not being searched thoroughly.

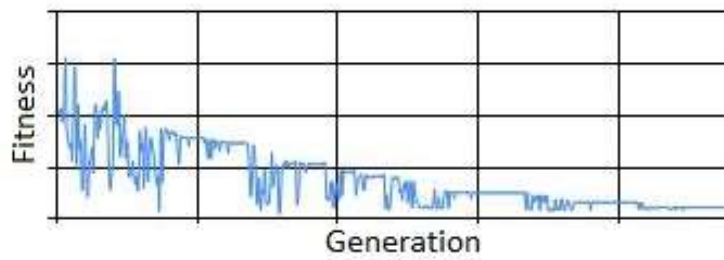


Fig. 5.3 – Mutation adjustment trial

After a few iterations, the final value of 4% of the mutation probability returned a much smoother graph (Figure 5.4). The problem with this configuration was the search time. Such a thorough search took 3 days to complete in the setup described on Section 5.3. At this point the focus shifted to the selection pressure, the last element left to be configured. If this parameter did not deliver a significant reduction of the search time, the entire configuration of the GA would need to start over with a different population size.

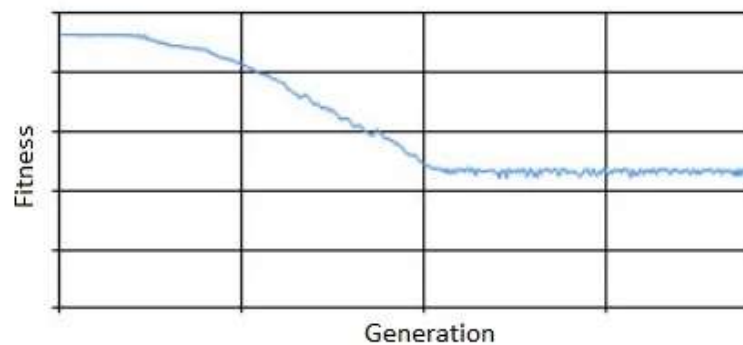


Fig. 5.4 – Ideal mutation probability reached

The selection pressure, is increased with the increase in size of the tournament pool as explained in Section 4.7. With a higher tournament pool, a low-quality individual has lower probability of being selected because there are more individuals he is compared with. In graphical terms, this results in a sharp increase in fitness of the initial generations (Figure 5.5). With a few iterations, it was found that a selection pool with 45 (20 % of total population) returned a good solution with a significant reduction of search time – down to 14 hours.

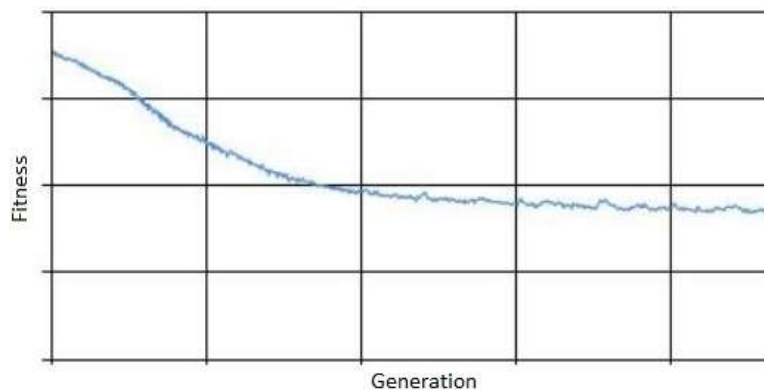


Fig. 5.5 – Final search

5.3. RESULTS

The optimization ran for 14 hours in a computer with 8GB of RAM and an Intel Core I7-6700HQ (8 threads) before the termination criteria of the genetic algorithm was met.

During runtime, several debug logs were collected and they revealed a bottleneck present in the Robot API. The function used to update bar properties for each individual evaluation was responsible for nearly 70% of the runtime of the entire optimization routine. Such delay in this operation, points to a possible limitation in the API to handle fast updates in models with several bars. This is a problem that will need to be addressed if the present software is to be used to optimize larger structures.

5.3.1. POSTPROCESSING

Given the steps taken to reduce the number of load cases, symmetrical load cases were removed. Only loads critical to the upper right quadrant, highlighted in Figure 5.6, of the tower were kept. This meant that the output would be a non-symmetrical structure where that upper right quadrant would need to be reproduced on the remaining three corners of the structure. In Figure 5.7 the critical quadrant is illustrated in isolation, next to the final symmetrical structure before any required strengthening work was performed.

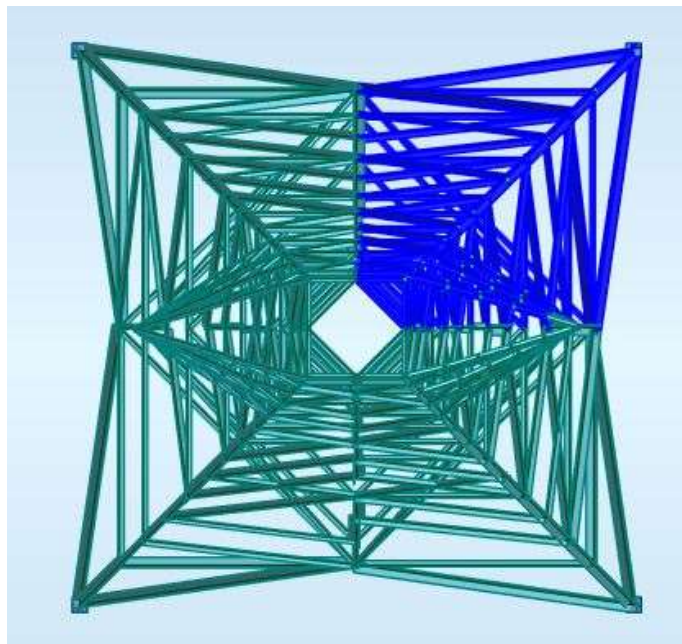


Fig. 5.6 – Plan view of the tower

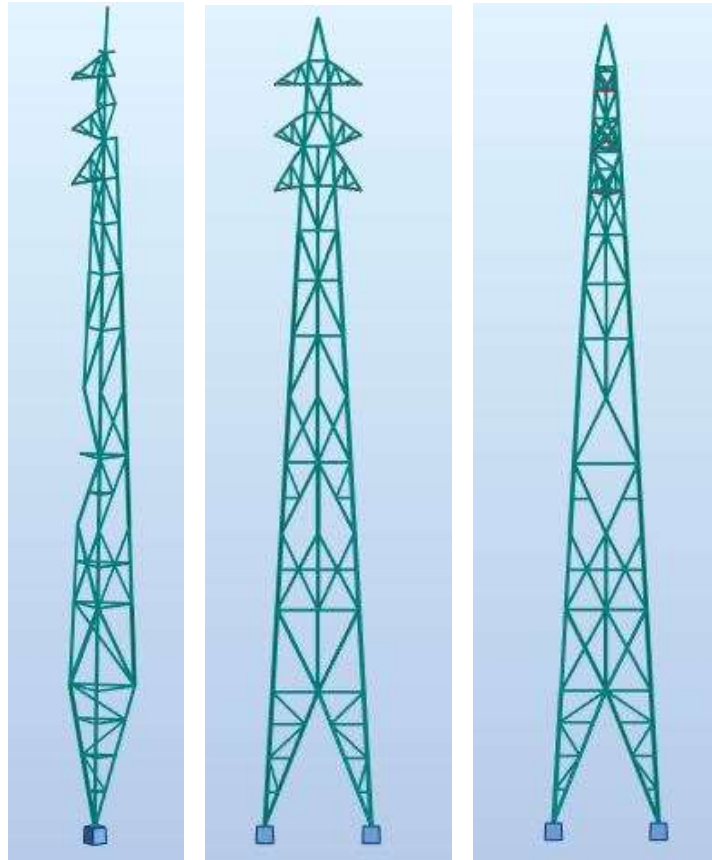


Fig. 5.7 – Critical quadrant, front and side planes

After the symmetry operations, the log file was opened to check if any bars needed additional strengthening. In this case, all the leg members were listed as well as a few bars in the middle of the structure with U/f slightly higher than 1.0. Given the locations of the elements to strengthen, secondary bracing, with section L 40x40x5, was used instead of larger sections.

Figure 5.8 details how the leg members were strengthened. A similar triangulation method was used for the other bars on the list.



Fig. 5.8 – Legs with secondary bracing added

5.3.2. ANALYSIS

The final structure has a total weight 6.8 tonnes, which corresponds to a 10.4% material reduction in comparison to the Metalgalva design. The weight is distributed between the different sections as follows:

Table 5.3 – Final structure weight distribution

Sections	Number	Total weight (kg)
L 40x40x5	72	330
L 50x50x5	281	1041
L 60x60x6	84	858
L 70x70x70	86	1482
L 100x100x10	55	3110
L 150x150x15	0	0
		6821

Adding to the material savings, the list of sections used is also smaller, allowing for more efficient manufacturing.

Given the current concerns about sustainability, savings will be analysed in terms of reduction of CO₂ emissions by using less steel. The national grid of the United Kingdom will be used as the data needed for the analysis is publicly available.

According to the European Strategic Energy Technologies Information System, in 2012 the European steel industry emitted 2.3 tonnes of CO₂ per tonne of steel. The report also identifies paths for improvement that could reduce this value by 70% to 0.7 tonnes of CO₂ per tonne of steel. For this analysis, the best-case scenario (the improvements were successfully applied to the European steel industry) will be used.

Data from the National Grid (UK) website points that there are 88000 electricity pylons in the UK and their average weight is 30 tonnes.

Using this information if a 10% reduction in material usage was applied to every pylon, 2 376 000 tonnes of CO₂ could be saved on the entire grid.

To make some sense of these numbers, data from the Environment Protection Agency (US) was used to compare these emissions with other activities that equate to similar emission levels, 2 376 000 is equivalent to:

- 501 891 Passenger vehicles (with average US yearly mileage) driven for one year;
- 11920 round trips to the moon in a passenger vehicle;
- 754 038 Tons of waste recycled instead of landfilled;
- 1 010 million litres of petrol burned;
- 600 installed wind turbines;
- Energy to power 250 000 homes for a year;
- 70 % of the yearly emissions of a coal-fired power plant.

5.4. CONCLUDING REMARKS

This first version of the program, validated the use of genetic algorithms to automate to a certain degree the design of optimised lattice tower structures, returning gains in productivity and material efficiency.

The returned structure also had fewer cross-section types than the original case study model. Such change is extremely relevant as it implies a reduction of material waste during the fabrication phase.

Having two different truss planes – symmetric in opposite faces – changes the fabrication phase as left and right sections of each bar need to be identified. According to Metalogalva, such a change is not a matter of concern as the fabrication phase is mostly automated. The assembly phase has a slight increase in complexity as each element needs to go not only to the correct place in the truss but also to the correct side of the structure.

5.4.1. FUTURE WORK

During development and testing, some points to improve in a future iteration of the software were also identified.

The Genome class, responsible for the DNA generation should be updated to ensure symmetry of the structure even when loads are reduced, as they were in the case study, to its most critical loads for a specific quadrant. This would shorten the execution time of the optimisation by reducing the number of load cases applied to the structure (by removing the symmetrical LCs). This avoids the need for human operations to re-establish symmetry of the solution. This could be implemented by defining a master quadrant, from which every bar in the remaining three corners of the structure would inherit its properties. The same approach should also be applied to the nodes.

When scaling up to optimise larger structures a constraint in the Robot API was identified. The calls needed to update bar properties between each individual evaluation, take too much time when compared to other calls such as the ones responsible for running and retrieving results from the structural analysis. In fact, when the bar count increases above a certain number, the communication between the developed program and Robot are slow enough to make the application stop responding. In a future version, changing the Robot_call class to work with another structural analysis package such as OpenSees or Oasys GSA will deliver higher performance and will enable more complex optimization problems to run.

Finally, to address the need for human intervention to read the log file and add secondary bracing to the model, a future academic work could study the implementation of a neural network to read input from that file and the structure, and automatically apply the needed bracing elements.

REFERENCES

Mckinsey & Company, *Imagining construction's digital future*. 2016.

<http://www.mckinsey.com/industries/capital-projects-and-infrastructure/our-insights/imagining-constructions-digital-future>, Accessed in: 12/06/2017

Brown MT, Bardi E. 2001. *Handbook of energy evaluation. A compendium of data energy computation issued in a series of folios. Folio 3: Energy of ecosystems*, Center for Environmental Policy, Environmental Engineering Sciences, University of Florida, Gainesville. 2001

Metalgalva, FEUP, 2015, *VHSSPOLES – Very High Strength Steel Poles*, Artes Gráficas, Porto

<https://www.mathworks.com/help/gads/some-genetic-algorithm-terminology.html> Accessed: 01/06/2017

Fogel, 2006. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE press, New York, USA

A. Lipowsky, 2011, *Roulette-wheel selection via stochastic acceptance*, Adam Mickiewicz University, Poznań, Poland

<https://www.autodesk.com/products/robot-structural-analysis/overview> Accessed: 01/03/2017

<http://www.rhino3d.com/> Accessed: 01/03/2017

<http://www.grasshopper3d.com/> Accessed: 01/03/2017

H.S. Bernardino, H.J.C Barbosa, A.C.C Lemonge , 2008, *A new hybrid AIS-GA for constrained optimization problems in mechanical engineering*, IEEE congress, Hong Kong, China

D. Orvosh; L. Davis, (1994), *Using a genetic algorithm to optimize problems with feasibility constraints*, IEEE conference, Florida, USA

<http://www.nashcoding.com/2010/07/07/evolutionary-algorithms-the-little-things-you-d-never-guess-part-1/> Accessed: 20/05/2017

<https://setis.ec.europa.eu/related-jrc-activities/jrc-setis-reports/energy-efficiency-iron-and-steel-industry-technology> Accessed: 06/06/2017