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 constrains (Figure 4.1 right) of the tower structure such as distance between ground supports, height of the power cables and number of arms.

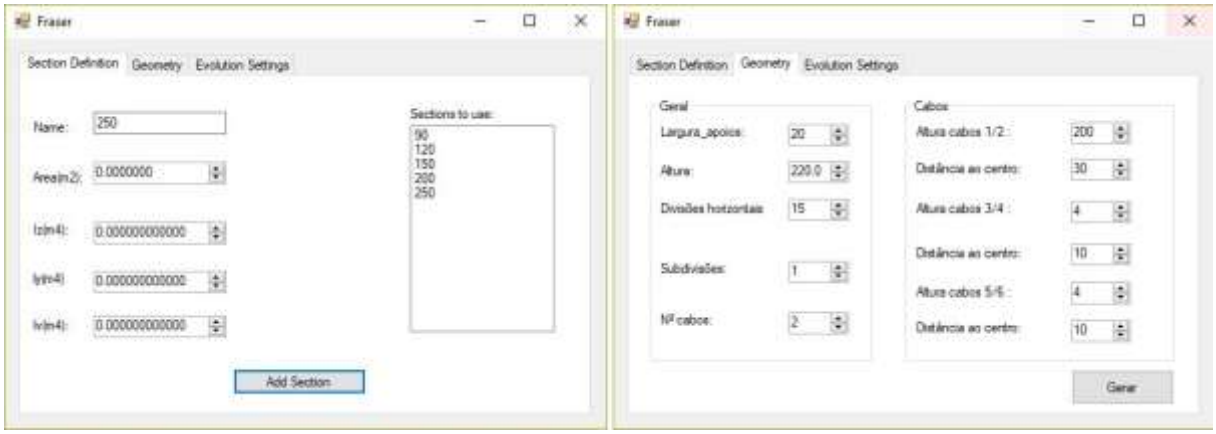

Fig. 4.1 – User interface

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 has 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.

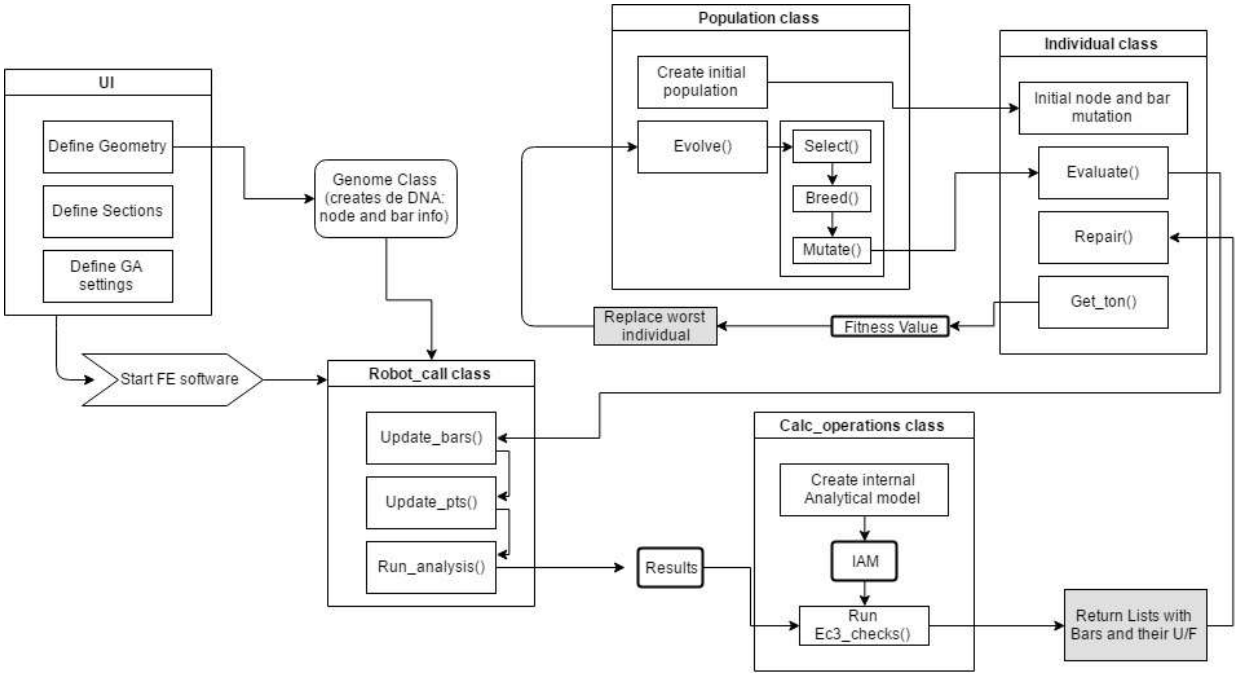

Fig. 4.2 – Interaction between key modules

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.

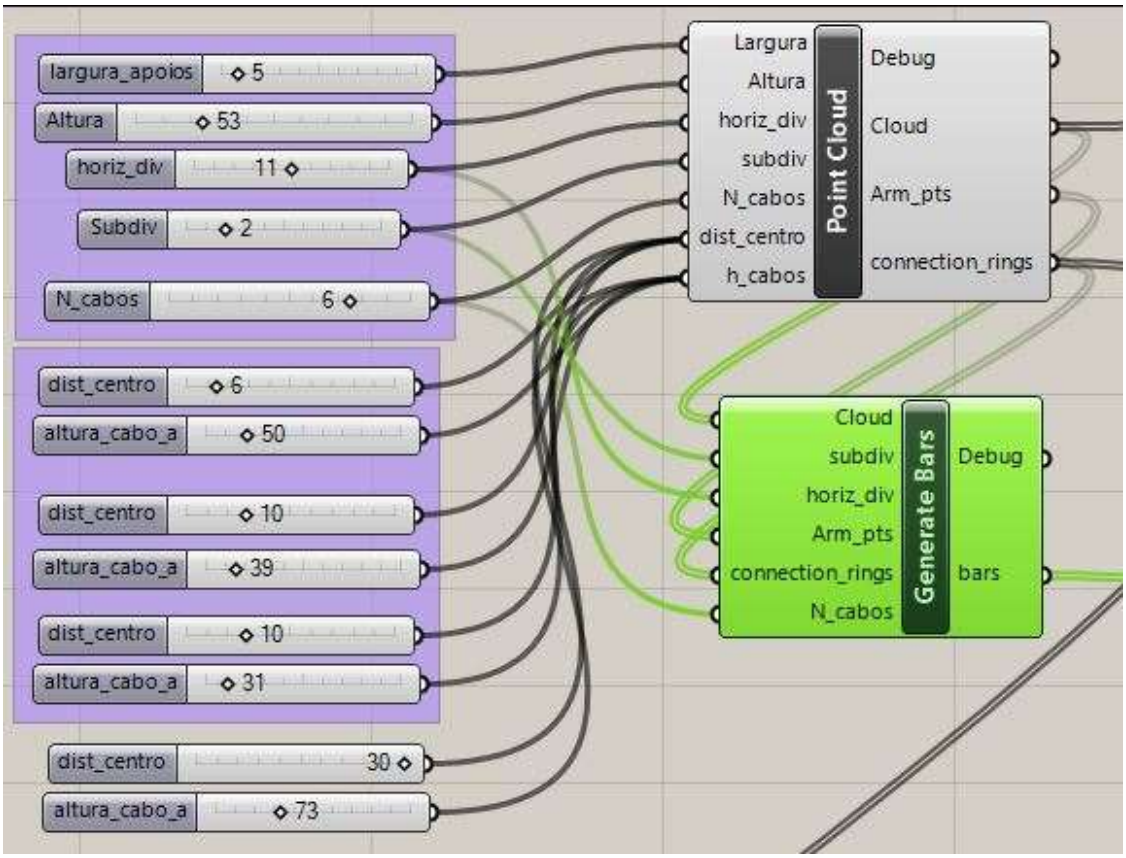

Fig. 4.3 Grasshopper

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.

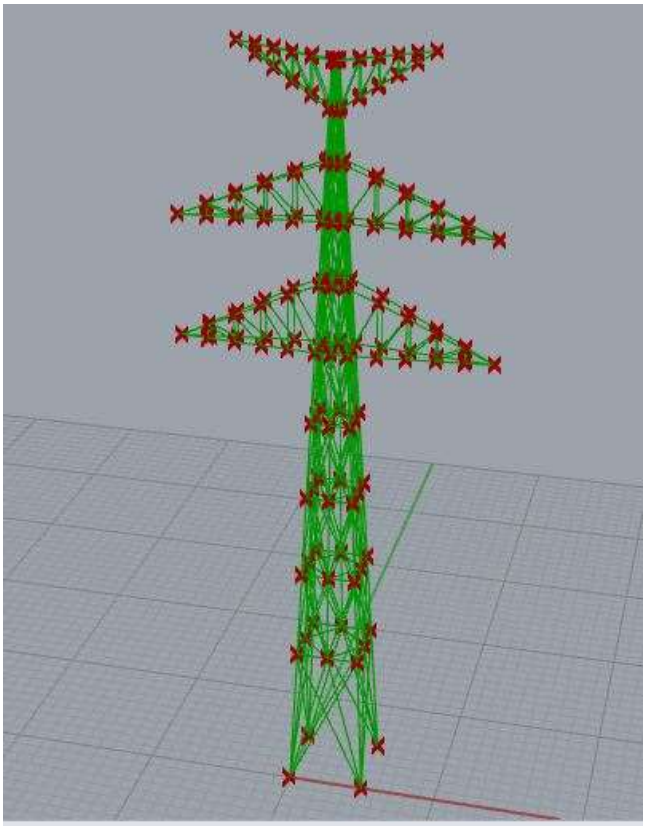

Fig. 4.4 – Grasshopper and Rhino model

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:

$$Nodes\text{: }[Number \mid x \mid y \mid z \mid mutation\ constant] \tag{4.1}$$

$$Bars\text{: }[Number \mid Point\ 1 \mid Point\ 2 \mid 1\ or\ 0 \mid Section \mid id] \tag{4.2}$$

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:

$$[initial\ bar|List\{unbraced\ real\ bars\}|\ buckling\ length] \tag{4.3}$$

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.

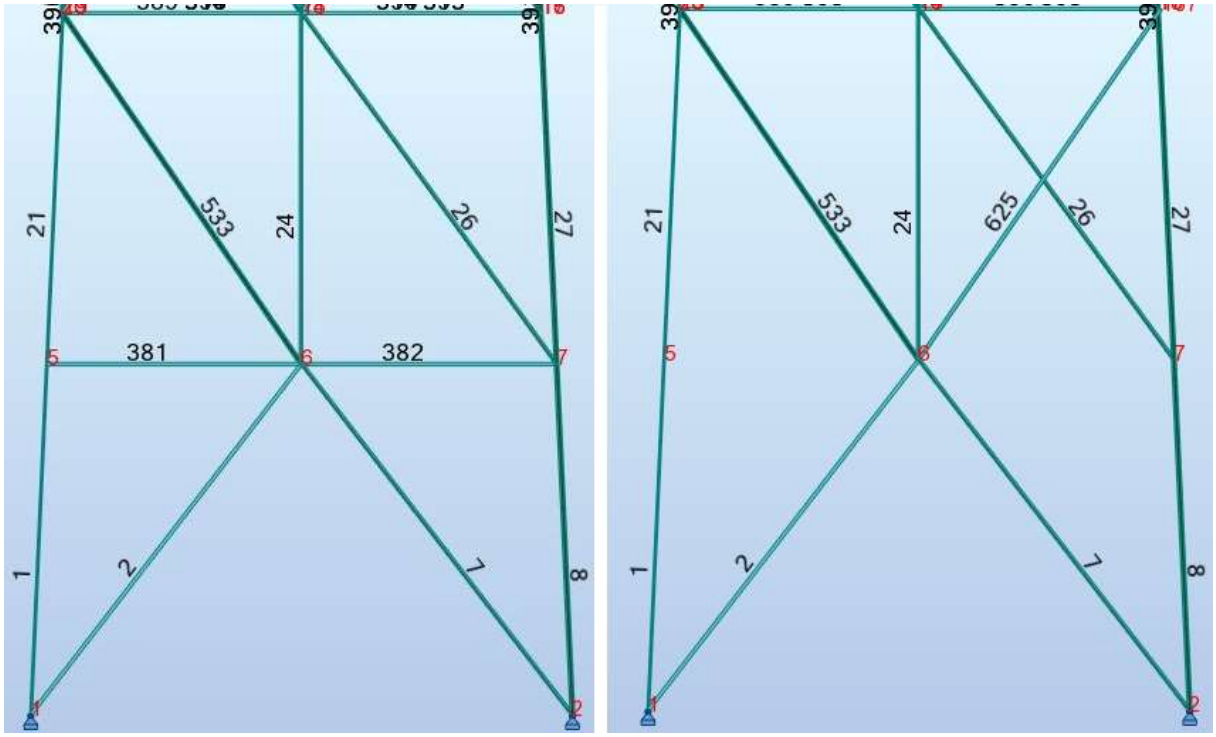

Fig. 4.5 – IAM: Different bracing conditions

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 overdesigned 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.

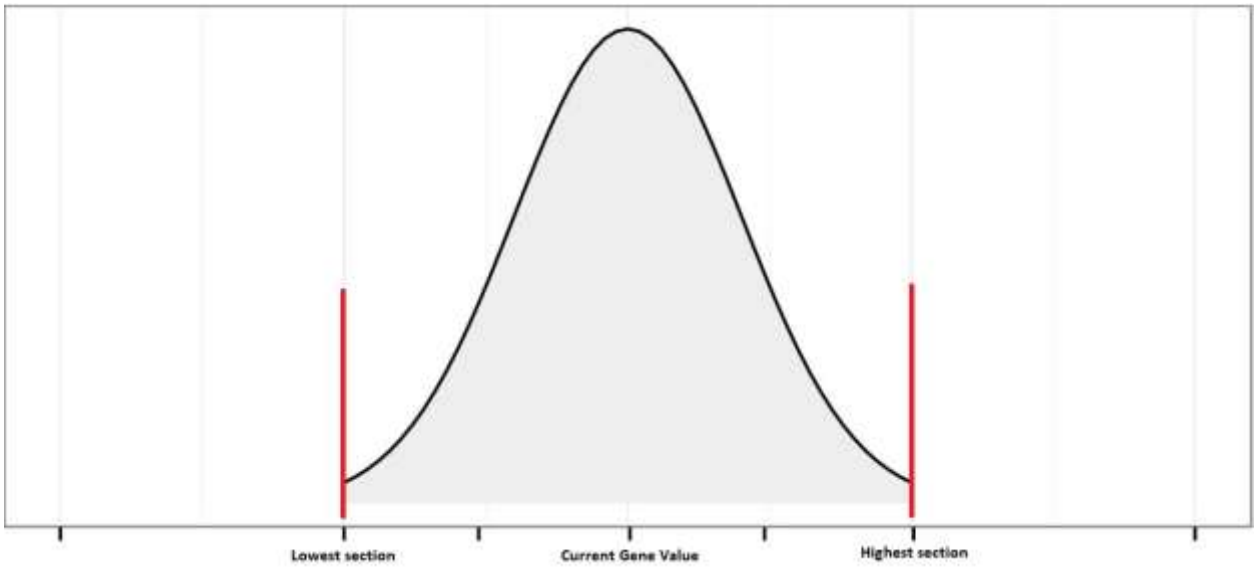

Fig. 4.6 – Tapered normal distribution

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; }
                }
```