**4**

**Program description**

**4.1. Summary**

In this chapter, the key elements 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 by the fitness function. For this program, the FEM software used was 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 the majority 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 where avoided allowing more time to be focused on the development of the GA and to solving technical problems. On the other hand 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, this process will be detailed in the next subchapters.

**4.2. Program flow**

In this subchapter, focus is given to the communications between different components and as such 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 tab where the used is asked to add properties of the sections the GA will used 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 to the tower structure such distance between ground supports, height of the power cables and number of arms.

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 (and realistic) bars active, to achieve that 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 in the Robot interface, those load cases will be applied to every individual in the population, the self-weight is automatically adjusted for each solution. With de setup process complete the initial population is created with a size that must be set by the user. For the initial population the x, y and z coordinates are randomly mutated and the section of each bar e randomly selected from all the sections defined and an extra section that translates to a disabled bar in the Robot model.

A FEM analysis is carried 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 type checking which bars are connected the end nodes and compiles that information in a calculation list that defines the internal analytical model that will be used for the Eurocode checks. That 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 off plane two entries to the list are added.

With the calculation list complete the fitness function is executed for each individual the Class “Calc\_operations” is responsible for running the EC3 checks and returning the 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. Bars in the first and third list add weight to the structure, the bars with low U/f already add weight in the real model by being overdesigned, for the bars that fail the EC3 checks a specific penalty function is defined.

The fitness value (in 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 operator 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.

[IMAGEM DOS VARIOS ELEMENTOS E SETINHAS]

**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 too liberal could expand the search space beyond what is acceptable with today’s computing power. (time????)

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 could decide which bars to delete and which coordinates to change.

To develop the algorithm that generates the base structure Rhinoceros and Grasshopper were used. Grasshopper is commonly known for its visual programming ability but 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. Using both programs helped streamline de development of the algorithm as any change showed on Rhino 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.

[Imagem do Grasshopper]

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

The node array stores the node number, cartesian coordinates and a mutation constant, this constant gives more control over the mutation in certain nodes where a normal mutation is not incremental enough, it effectively scales down the mutation in 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, that behaviour could prevent the GA from reaching the optimum as each mutation would overshoot the target.

The bar array defines the bar number, the start and end nodes, the fourth column receives a 1 or 0 if the bar can be deactivated or not – Leg and arm members cannot be deactivated – and the fifth column is and 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.

|  |
| --- |
| 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++;  }  }  (…) |

[Adicionar também a imagem do grasshopper -> robot]

**4.4. Initial population**

The initial population is created when the user defines the initial geometry, its size is also set 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 sections to try doesn’t need the same number of initial random individuals to adequately populate the solution space as a highly complex tower with 1000s of bars and 10s sections to try.

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

|  |
| --- |
| 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; //a primeira secção a definir será a dos braços  //secção de braços  }else  {  this.\_DNA.bars[4, i] = Population.rand.Next(0,Sections.count-1); // rand. section  } |

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

|  |
| --- |
| 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();  c++;  series.Points.AddXY(c, CurrentPop.ind[a].fitness); //Plot graph  }  }  (…) |

**4.5. Evaluate function**

This function is responsible for the majority of the intensive calculations that are carried 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 it 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 and 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.

|  |
| --- |
| 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 follow the same general workflow: The algorithm runs through adjacent bars of the same type and at each intersection with other bars it checks to see if that type of intersection provides bracing, if it does a list entry is added with a new analytical bar number and the real bars that makeup that bar, from that information the Buckling length is added. The function also checks for different buckling lengths off plane and in plane. The following image explains how the algorithm stores a leg element in the IAM.

[meter imagem exemplo de um elemento LEG]

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

The Eurocode check have already been described on chapter 2, the following code is just the verification translated into code for the LEG elements, similar code is present in the EC3\_checks function for other bar types.

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

The three lists are used by the fitness function to add penalties to over or under designed bars, the penalty function is as follows [formula] and the correspondent code snippet is:

|  |
| --- |
| ///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) // se ainda não estiver com a maior secção pode aumentar  {  Console.WriteLine("Poder aumentar Sec da barra" + temp[0]);  this.fitness += 5; // penalização de 5 ton  }  else  {  this.fitness += 10;  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.

**4.6. Selection function**

For the selection function the tournament selection was used after tests with the roulette wheel selection revealed the lack of adjustability the algorithm provided, being it a minimization problem the implementation of the tournament selection was also more straight forward.

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 is the implementation of the tournament selection described.

|  |
| --- |
| 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>()];  } |

**4.7. Genetic operators**

4.7.1. Crossover

The single point crossover was tested in a previous version of the code but it did not help the convergence of the algorithm, in complex structures the search turned 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 is the crossover function applied to the bar array, the implementation is similar for the node array but the changed values are x,y,and z coordinates.

|  |
| --- |
| ///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 initial takes all the genes from the best individual  ///parts of the chromossome are changed for the other selected individual (b) with  ///the following code:  x.\_DNA.bars[4, i] = b.\_DNA.bars[4, i]; ///Change section  } |

4.7.2. 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, and it was valuable information for the mutation function implementation.

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 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 less than 10cm and 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 selection the next value for the gene value from a Gaussian distribution centred on the current value of the gene and with a used defined sigma value.

[imagem]

The implementation is as follows: [ADD credits!!]

|  |
| --- |
| private static double gaussianMutation(double mean, double stddev)  {  double x1 = rand.NextDouble();  double x2 = rand.NextDouble();  // The method requires sampling from a uniform random of (0,1]  // but Random.NextDouble() returns a sample of [0,1).  // Thanks to Colin Green for catching this.  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) and extra function is called to correct values that fall outside that range:

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

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

|  |
| --- |
| if (\_rnd < sec\_mutation\_prob)  {  if (x.\_DNA.bars[3, i] == 1) // se pode ser desactivada  {  double sigma = Sections.count / 2;  double gene\_val = gaussianMutation(x.\_DNA.bars[4, i], sigma);  gene\_val = clamp(gene\_val, -1, Sections.count - 1); // min val = -1 ==== disable section  //x.\_DNA.bars[4, i] = Population.rand.Next(0, Sections.count - 1); //se pode ser descativada random de 0 ate sec count  x.\_DNA.bars[4, i] = (int)gene\_val;  cnt++;  } else // se nao pode ser desactivada  {  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); // min val=0 === min sec  // x.\_DNA.bars[4, i] = Population.rand.Next(1, Sections.count - 1); // se nao random de 1 ate sec count  x.\_DNA.bars[4, i] = (int)gene\_val;  cnt++;  }  if(x.\_DNA.bars[5,i] == 5) { x.\_DNA.bars[4, i] = 0; } // barras trianguladas intra plane sao sempre sec minima  } |