Optimization of the metamorphic development process via the genetic algorithm

Keith Daniel Joseph Goren
Rowan University

Follow this and additional works at: http://rdw.rowan.edu/etd
Part of the Mechanical Engineering Commons

Recommended Citation
http://rdw.rowan.edu/etd/727

This Thesis is brought to you for free and open access by Rowan Digital Works. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Rowan Digital Works. For more information, please contact LibraryTheses@rowan.edu.
OPTIMIZATION OF THE METAMORPHIC DEVELOPMENT
PROCESS VIA THE GENETIC ALGORITHM

by
Keith Daniel Joseph Goren

A Thesis
Submitted in partial fulfillment of the requirements of the
Masters of Science Degree
of
The Graduate School
at
Rowan University
August 24, 2007

Approved by
Date Approved 2/3/08
Advisor: Dr. Eric Constans

Approved by
Date Approved 3/4/08
Committee Member: Dr. VonLockette

Approved by
Date Approved 3/4/08
Committee Member: Dr. Chandrupatla

© August 24, 2007 Keith Daniel Joseph Goren
ABSTRACT

Keith D. J. Goren
OPTIMIZATION OF THE METAMORPHIC DEVELOPMENT PROCESS VIA THE GENETIC ALGORITHM
2007/08
Dr. Eric Constans
Masters of Science in Mechanical Engineering

This main investigation of this thesis was the integration of the metamorphic development process with a genetic algorithm in order to optimize the metamorphic development process. Since the origin of the metamorphic development process, there has been a need for the user to input parameters in order to coax structural development to obtain an optimum structure. By utilizing the genetic algorithm to manipulate this input information, this new marriage of the metamorphic development process and the genetic algorithm can save time and computation for the user of the program. It was found that the genetic algorithm with the metamorphic development process can obtain an optimized solution for a given structural environment. A second dimension of this thesis, lesser in importance yet still notable, is the use of a non-grid based finite element mesh.
Acknowledgements

First, I thank the Lord for giving me the gifts and talents necessary to accomplish all that I have done. Without Him I can do nothing.

Second, my thesis advisor, Dr. Constans, who has helped guide this project from its beginning. His supervisions of this thesis has truly made it what it is today.

Finally, I would like to thank my family, my beloved wife Lisa and son Jeremy. If they did not push me to finish this thesis, it may never have been completed.
TABLE OF CONTENTS

ACKNOWLEDGEMENTS........................................................................... ii

LIST OF FIGURES.................................................................................. v

LIST OF TABLES..................................................................................... vii

CHAPTER 1: OVERVIEW AND INTRODUCTION........................................... 1

INTRODUCTION..................................................................................... 1
OBJECTIVES.......................................................................................... 2
PRIOR WORKS CITED.......................................................................... 2
DIFFERENTIATION FROM PREVIOUS WORK...................................... 5

CHAPTER 2: THE HISTORY OF THE MD PROCESS................................. 7

A.G.M. MICHELL.................................................................................. 7
THE ESO METHOD.............................................................................. 8
THE BESO METHOD........................................................................... 9
THE MD PROCESS............................................................................. 10
OVERVIEW OF THE NEXT CHAPTERS............................................. 11
VARIABLE FLOWCHART...................................................................... 12

CHAPTER 3: THE METAMORPHIC DEVELOPMENT PROCESS................ 14

ROOTING IN ORGANIC DEVELOPMENT............................................. 14
THE MD PROCESS IN MATLAB......................................................... 14
GROWING NEW CELLS....................................................................... 16
EXPANSION OF EXISTING CELLS..................................................... 23
ELIMINATION OF UNNECESSARY CELLS........................................ 25

CHAPTER 4: OPTIMIZATION VIA THE GENETIC ALGORITHM............... 30

OPTIMIZATION TECHNIQUES IN GENERAL....................................... 30
ADVANTAGES OF THE GENETIC ALGORITHM FOR OUR PURPOSES... 30
VARIABLE AND CONSTANT DEFINITION.......................................... 31
THE INITIAL POPULATION.................................................................. 32
CONVERTING FROM BINARY GENES TO REAL WORLD INFORMATION.......................... 33
CHECKING FOR CONVERGENCE........................................................................ 34
GOAL OF OPTIMIZATION FOR THIS THESIS.................................................. 34
DEVELOPING THE NEXT GENERATION OF BRIDGES........................................ 35

CHAPTER 5: OPTIMIZATION OF THE METAMORPHIC DEVELOPMENT PROCESS VIA THE GENETIC ALGORITHM..................... 39
THE GENERIC GENETIC ALGORITHM............................................................. 39
THE NEED FOR A GENETIC ALGORITHM...................................................... 39
FLOW OF CONTROL IN THE GENETIC ALGORITHM...................................... 40
EVALUATION OF EACH BRIDGE..................................................................... 41

CHAPTER 6: RESULTS AND CONCLUSIONS................................................. 43
GOAL OF THE THESIS...................................................................................... 43
EFFECT OF THE GROWTH FACTOR................................................................ 43
CONVERGENCE............................................................................................... 44
PRELIMINARY STRUCTURE IMAGES.............................................................. 45
THE CONVERGED ANSWER............................................................................ 46
FINAL CONCLUSIONS ABOUT SIMPLY SUPPORTED STRUCTURE.................. 48
FUTURE WORK................................................................................................. 53
REVIEW OF THIS THESIS' WORK................................................................. 53

REFERENCES.................................................................................................. 55
APPENDICES.................................................................................................... 57

APPENDIX A: MATLAB CODE........................................................................ 57
APPENDIX B: CONVERGENCE OF MULTIPLE STRUCTURES THROUGH SEVERAL GENERATIONS......................................................... 94
### List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1-1</td>
<td>6</td>
</tr>
<tr>
<td>Figure 2-1</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2-2</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2-3</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2-4</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2-5</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2-6</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2-7</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2-8</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3-1</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3-2</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3-3(a)</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3-3(b)</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3-4</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3-5</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3-6</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3-7</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3-8</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3-9</td>
<td>25</td>
</tr>
<tr>
<td>Figure 3-10</td>
<td>29</td>
</tr>
<tr>
<td>Figure 4-1</td>
<td>33</td>
</tr>
<tr>
<td>Figure 4-2</td>
<td>36</td>
</tr>
<tr>
<td>Figure 4-3</td>
<td>Dart Strip Roulette Wheel</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>Figure 4-4</td>
<td>Crossover at Two Points</td>
</tr>
<tr>
<td>Figure 5-1</td>
<td>The Genetic Algorithm with Metamorphic Development</td>
</tr>
<tr>
<td>Figure 6-1</td>
<td>The Growth Effect</td>
</tr>
<tr>
<td>Figure 6-2</td>
<td>High Factor, GrowStress and Low NewStress</td>
</tr>
<tr>
<td>Figure 6-3</td>
<td>Low Factor and High KillStress</td>
</tr>
<tr>
<td>Figure 6-4</td>
<td>The Converged Solution</td>
</tr>
<tr>
<td>Figure 6-5</td>
<td>Arch with Load Unconstrained</td>
</tr>
<tr>
<td>Figure 6-6</td>
<td>Straight Bar and M-ing Model Stresses</td>
</tr>
<tr>
<td>Figure 6-7</td>
<td>Archway Compression Lines</td>
</tr>
<tr>
<td>Figure 6-8</td>
<td>M-ing Phenomenon Silhouette</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3-1</td>
<td>16</td>
</tr>
<tr>
<td>Table 6-1</td>
<td>44</td>
</tr>
<tr>
<td>Table 6-2</td>
<td>45</td>
</tr>
</tbody>
</table>

- Table 3-1: List of Variables and Structures
- Table 6-1: Generation One Data
- Table 6-2: Generation Twenty Data
CHAPTER 1: OVERVIEW AND INTRODUCTION

INTRODUCTION

Many researchers in the past century have contributed to the advancement of topology optimization. Starting with A.G.M. Michell in 1904 [6] up to present contributors such as Jing-Sheng Liu, Geoff Parks and John Clarkson [4, 5], many techniques have been developed which solve for an optimum structure given a set of constraints and conditions. In the past decade, a new concept in topology optimization has emerged, the metamorphic development process. This process utilizes the finite element method and many other techniques to determine the optimum topology of a structure given a certain set of constraints and loads on the initial structure. Because it is the one of the most updated method in the field of topology optimization, it will be one of the basic techniques used for the research of this thesis. The primary drawback of this method is its heavy dependence on the specific variables input by the programmer working together to find the optimum solution. If all of the variables do not work together, the solution to the set of constraints will be suboptimal and therefore useless to the person using the metamorphic development process.

The other method being used is a more common idea in the scientific community, that of the genetic algorithm. The genetic algorithm, more commonly known as ‘survival of the fittest’, is a common optimization technique used in a variety of different mathematical and engineering applications [2]. The genetic algorithm will be the foundation for the research in this thesis, as it will aid us to evolve the metamorphic development process. This thesis will use the
genetic algorithm to optimize variables that have a primary impact on the metamorphic development process.

**OBJECTIVES**

The goal of this thesis is the integration of the two basic methods as mentioned in the introduction, namely the metamorphic development process and the genetic algorithm. The user will be able to allow the algorithm to determine the variables which impact the metamorphic development process rather than relying on the user to input the correct variables and hope they all work together. The success of this method will relieve the user from having to guess an undetermined amount of times at a set of variables which will yield a plausible answer to the topology problem at hand.

A secondary objective of this thesis is to allow the development process more freedom to acquire the optimum solution by unrestricting the nodes from fixed locations. A variety of nodes create a mesh of points which will define the shape of the final topology. If these nodes are restricted, our final topology will also be restricted. Although this is a secondary objective, it is nonetheless a significant development in the evolution of the metamorphic development process.

**PRIOR WORKS CITED**

As in the case of any new development in science, engineering or technology, there is a set of background sources and information that must be credited. Listed below are a variety of different sources used to fuel the basis of
this thesis. Among these sources there are two books, a variety of published articles and some online aids.

The first two sources, both published articles, contributed indispensably to the metamorphic development portion of this thesis. The first article, *Metamorphic Development: A New Topology Optimization Method for Truss Structures* \([4]\), written by Jing-Sheng Liu, Geoff Parks and John Clarkson at the University of Cambridge in 1999, has been one of the most recent developments in topology optimization. As seen in chapter two, the metamorphic development process is the most recent development in a long series of different truss optimization techniques developed over the last century. This article is the first presentation of the metamorphic development process to the world, and therefore it carries with it the most raw information concerning this process, but also can be developed upon tremendously, as even the authors mention in the introduction. Our goal is to accomplish just that, to expand upon their framework for the metamorphic development process.

The second article, *Metamorphic Development: A New Topology Optimization for Continuum Structures* \([5]\), written by the same authors, Jing-Sheng Liu, Geoff Parks and John Clarkson, covers similar material as the previously mentioned article. In the previously mentioned article, the finite elements used were strictly limited to truss elements. In the current article, the process expands its capacity to include bar, or square, elements as well as the triangular elements. It also helps develop and explain the intricacies of the metamorphic development process.
The first book that was used for this thesis' research is entitled - *Introduction to Finite Elements in Engineering* [1], co-written by Tirupathi R. Chandrupatla and Ashok D. Belegundu. It is one of the world's leading books in the field of finite element theory and was instrumental in the aid of the programming for the finite element dimension of this thesis. Although finite element theory is well known in the engineering community at this point, this book was still important to develop a better understanding of the metamorphic development process and the construction of the programming (with the help of T.R. Chandrupatla in guiding some of the use of the programming code from the book).

The other book, also co-written by Tirupathi R. Chandrupatla and Ashok D. Belegundu, is *Optimization Concepts and Applications in Engineering* [2]. This book was not used as extensively as the one mentioned earlier, but nonetheless was useful in the basic understanding of optimization techniques. In this book, section 7.8 focuses its attention on the genetic algorithm. Although the genetic algorithm is a well known concept in the engineering community it was still necessary to use a basic outline for the generation of the program. The outline seen in section 7.8 of this book is the same generic outline for the genetic algorithm that is described in much further detail in chapter 4 of this thesis.

Another source used was *An Adaptive Penalty Function in Genetic Algorithms for Structural Design Optimization* [8] written by Pruettha Nanakorn and Konlakarn Meesonklin. This article helped shape the penalty function dimension of this thesis. They state, "The inconveniences of this technique (genetic algorithm) are how to choose the initial value for the penalty coefficient
and how to appropriately update it." At the end of chapter 5 of this thesis the penalty function will be discussed and how these issues have been treated.

In *Combining Approximation Concepts with Genetic Algorithm-Based Structural Optimization Techniques* [7] written by P.B. Nair and A.J. Keane from the University of Southampton, the idea of tradeoff between robustness and efficiency is discussed with regard to the genetic algorithm process. They also mention the unfortunate situation where a suboptimal solution can be found when the program a local optimum solution is obtained in place of a global optimum. In this thesis, the roulette method has been used, which is discussed at length in chapter 4 under the subsection ‘Developing the Next Generation of Structures’.

*Modern Structural Optimization Concepts Applied to Topology Optimization* [3], written by Juan Pablo Leiva, Brian C. Watson, and Iku Kosaka from VMA Engineering, discusses concepts that will be useful in the continued research of this project. They discuss the need for the future of topology optimization to allow for the program to determine the different material properties rather than initializing the program to use only specified materials. This concept will be further discussed in chapter 6 in future suggestion for this project.

**DIFFERENTIATION FROM PREVIOUS WORK**

Unlike all of its predecessors, the version of the metamorphic development algorithm developed here is not fixed to a grid. When the program used is grid based, the computer programming code is much easier, but does not allow for the
structure to have smooth features. Either the mesh of elements needs to be very fine or the programmer must spline-curve fit. Figure 1-1 shows two examples of spline-curve fitting using different sized elements. The blue, solid section is filled to smooth the edges. One of the basic principles of structural mechanics is that sharp edges have high concentrations of stress. This cannot be overcome by the program; therefore the program user must artificially smooth the resulting structures.

One unique dimension of this project is the optimization technique's role in relation to the larger scale of the thesis. In most cases, the optimization technique directly solves the problem at hand. In this case, however, the optimization technique, the genetic algorithm, focuses on optimizing the inputs or parameters, to the metamorphic development process, which is ultimately solving the question, 'Which structure will be the optimum solution?'.
CHAPTER 2: THE HISTORY OF THE MD PROCESS

The metamorphic development process has undergone considerable evolution over the past decade. Through the efforts of different people in the twentieth and twenty-first centuries, the process has come to be a useful tool in the field of topology optimization as the reader will see throughout this chapter.

A.G.M. MICHELL

In 1904, A.G.M. Michell wrote *The Limits of Economy of Material in Frame-structures* [6]. In his book he was the first to write on the importance of the relationship between the mass of a structure and its structural integrity. He saw that it was not only important to design a structure that met the physical requirements of a situation, but also to minimize the amount of material used to accomplish this task. He used two primary examples when modeling this concept, the same two basic setups that are used as benchmarks today. First, the cantilever/wrench example is used. This example has a ring of constraints on one side of the structure and a force exerted perpendicular to the structure on the opposite side. In the optimal solution, the structure is tear-shaped with the point of the drop at the force, as seen in Figure 2-1. The other example of a Michell structure is the simply supported beam. The two bottom corners of the structure are supported in the y direction and one corner is supported in the x direction. The force is exerted in the center of the bottom of the structure in the negative y direction. The structure in Figure 2-2 shows how it is optimized and forms a web
of material. Both of these structures display the minimum use of material with respect to the mass and structural integrity.

THE ESO METHOD

Ninety years later, in the time when computers became a viable means to solve computationally expensive problems, the Evolutionary Structural Optimization (ESO) technique was developed by X.Y. Yang, Y.M. Xie, G.P. Steven and O.M. Querin [9]. Based on the finite element method, the ESO takes a filled grid of elements and begins to evaluate them. This alpha version takes the filled grid and eliminates the elements that have very little stress on them. There is no ability for the elements to regenerate or grow, but only to eliminate the unnecessary element. This was a good concept to begin with, but it also has many shortcomings. First, this method is very computationally intensive. By filling the grid at the outset, the program must compute each finite element at each iteration. Second, the method does not allow growth, so if an element is removed it does not have the opportunity to grow back. This leaves a crude result from the initial mesh structure, as seen in Figure 2-3. Finally, the method based on a grid, and therefore the mesh elements are restrained to the points on the grid alone.
Six years later, in 1999, X.Y. Yang, Y.M. Xie, G.P. Steven and O.M. Querin developed a higher level ESO, BESO [9]. The Bidirectional Evolutionary Structural Optimization improves on its predecessor in two ways. First, the grid does not have to be filled with elements at the outset of the program. Instead, the program needs a relatively basic structure that has the shape of the final product. Second, as the Bidirectional part of BESO implies, this method has the ability to grow elements as well as remove them. If the stress of an element is very low, the element is removed. If the stress is higher than some maximum stress, the element in question grows another element adjacent to itself. Although superior to the ESO method, the BESO method still incurs some major faults. Again, it is computationally expensive because the method begins with a relatively large structure. Also, if the initial structure is not a somewhat good representation of the final structure the answer will not reach maturity and will be suboptimal. Finally, this method also is based on a preset grid. The structural elements cannot leave the grid.
Figure 2-4 shows the initial structure and results for a BESO model with cantilever constraints.

THE METAMORPHIC DEVELOPMENT PROCESS

In 1999, J.S. Liu, G.T. Parks and P.J. Clarkson created the metamorphic development process \cite{4, 5}. Similar to its older predecessor, the MD process has a similar method, but different parameters. First, the MD process can start as a very simple structure that is not a necessarily close representation to the final product. This allows the user the convenience of not trying to guess at the answer before it is developed. Also, this allows poor initial guesses at the structure the freedom to become good answers. Secondly, the growth and decay of the elements is not based on a single constraint. The constraint for the growth and decay of elements changes functionally with the number of generations the structure has undergone development. This allows the structure to develop "organically" through growth and decay, in the manner of a living organism. The growth process can be more active in the beginning of the life of the structure and the decay process can become more stringent towards the end of the life of the structure. There are a variety of different functions that can be used to effect this, but the two primary functions used are the basic linear change and the arctangent.
change, seen in Figures 2-5 and 2-6 respectively. The blue, dashed lines in both figures are the stress needed at the current year to make a new element. The red, solid lines are the stress needed to kill an element at the current year. This is further explained in the section dedicated to the metamorphic development process.

OVERVIEW OF THE NEXT CHAPTERS

Now that the history of the metamorphic development process has been discussed, we will begin to discuss the present form of the process. In the next chapter, the metamorphic development process will be explained in detail as it is used in this thesis.

To give the reader an overview of the following chapters, the flowchart in Figure 2-7 is provided. Chapter 4 will cover the genetic algorithm, which is white in the chart. Chapter 3 will cover the metamorphic development process, which is dark gray in the flowchart. Chapter 5 will discuss how the two processes interact, shown in light gray. Finally, Chapter 6 will reveal the findings and conclusion of this thesis.
VARIABLE FLOWCHART

The following chapters explain the functions of a variety of different variables. In Figure 2-8, the flow of the variables between the genetic algorithm and the metamorphic development process can be seen in the arrows. Table 3-1 gives a clear definition of the variables. Figure 2-7 and Figure 2-8 are identical in process, but show different information.
Figure 2-8: Flowchart of Variables
(see Table 3-1 for variable explanations)
In the development of each individual in a population, or grouping of individuals, there are many stages that must be accounted for, just as in the development of any living organism. The stages of development are the same for each individual, but based on the genetic input, which controls the individual's growth and decay. The final shape of each individual is unique. There are three developmental stages that each structure undergoes in every year of its life. First, a structure can grow by adding cells to open spots on the perimeter of the existing structure. This is similar, but not identical, to cellular mitosis as seen in Figure 3-1. Next, each cell has the opportunity to grow, or enlarge. This process does not change the number of cells, but the area of individual cells. Finally, each cell's worth is evaluated and the cells that are unnecessary are eliminated. This can be compared to the atrophy of cells in an unused muscle.

THE MD PROCESS IN MATLAB

Throughout the metamorphic development process, a variety of different structures and variables will be mentioned. This section will familiarize the
reader with the structures and variables used throughout this chapter.

Primarily there are three structures that need to be presented: *Elem, Edge* and *Node*. The *Elem* structure contains all of the information needed to represent all of the elements in the current individuals. The structure holds information regarding the nodes and edges associated with the elements, the stress within each element, the material for each element, the area of the element and the age of the element. Figure 3-2(a) shows an individual with all of the elements numbered. The *Edge* structure contains information about the edges in the individual. The *Edge* structure indicates what two nodes are associated with an edge, what element(s) are associated with an edge, and whether or not a new element can be grown off of an edge. Figure 3-2(b) shows an individual with all of the edges numbered. Finally, *Node* is an array that contains the x and y coordinates of each node in the individual. Figure 3-2(c) shows an individual with all of the nodes numbered.

**LIST OF VARIABLES AND STRUCTURES**

In the following list, the variable appears on the left, the name used in the code appears in the center, and if necessary a short explanation appears on the right.

<table>
<thead>
<tr>
<th>$S_N$</th>
<th>$NewStress$</th>
<th>Stress required for a new element to be created</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{Nmax}$</td>
<td>$NewStressMax$</td>
<td>Approximate stress required for a new element to be created in the final year</td>
</tr>
<tr>
<td>$S_G$</td>
<td>$GrowStress$</td>
<td>Stress required for an element to grow</td>
</tr>
<tr>
<td>$S_{Gmax}$</td>
<td>$GrowStressMax$</td>
<td>Approximate stress required for an element to grow in the final year</td>
</tr>
<tr>
<td>$S_R$</td>
<td>$RemStress$</td>
<td>Stress required for an element to be removed</td>
</tr>
<tr>
<td>$S_{R\text{max}}$</td>
<td>$\text{RemStressMax}$</td>
<td>Stress for an element to be removed in the final year</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$N$</td>
<td>$\text{DeathYear}$</td>
<td>Total lifespan of an individual</td>
</tr>
<tr>
<td>$n$</td>
<td>$\text{Year}$</td>
<td>The current year in the life of an individual</td>
</tr>
<tr>
<td>$i$</td>
<td>$\text{Individual}$</td>
<td>One person in the population</td>
</tr>
<tr>
<td>$j$</td>
<td>$\text{Element}$</td>
<td>One element in an individual</td>
</tr>
<tr>
<td>$dx$</td>
<td>$\text{Deltax}$</td>
<td>The distance between two $x$ locations</td>
</tr>
<tr>
<td>$dy$</td>
<td>$\text{Deltay}$</td>
<td>The distance between two $y$ locations</td>
</tr>
</tbody>
</table>

Table 3-1: List of Variables and Structures

GROWING NEW CELLS

JUSTIFICATION FOR NEW CELLS

The function NewElem.m, seen in the appendix, is the subroutine that creates new elements when necessary. The inputs into this subroutine are the nodes, elements, edges, constraints, loads, material properties, age of death for the individual, current age of the individual and maximum stress that the cells can endure before needing to multiply. There are a variety of subroutines called from the routine NewElem.m, each having a specific purpose in the process of

![Figure 3-2: Elements, Edges and Nodes (Disregard color gradient)](image-url)
making a new element, which will be explained in detail in this section.

NEW STRESS MAXIMUM VALUE

First, the routine initializes a variety of variables. The most important variable in this routine is $S_N$, the variable that determines if a cell is under enough stress to grow another element adjacent to it. The $S_{N\text{max}}$ variable is an input from the genetic algorithm, and is one of the variables being optimized in the overall program. Because each structure is a growing and developing organism, it would be inappropriate to set one stress level for its entire life for each cell to meet or exceed. The growth process can be more active in the beginning of the life of the individual. There are a variety of different functions that can be used to effect this, but the two primary functions used are the basic linear change and the arctangent change, seen in Figures 3-3(a) and 3-3(b) respectively. The blue, dashed lines in both figures are the stress values needed at the current year to make a new element. The red, solid lines are the stress levels needed to kill an element at the current year (which will be discussed later in this chapter).

It was decided to use a hyperbolic tangent curve to determine what the stress level was at each year of an individual’s lifespan. The formula for $S_N$ is:

$$S_N = \left[ \tanh \left( \frac{8n}{N} - 4 \right) + 1 \right] \left[ \frac{S_{N\text{max}}}{2} \right]$$

where $N$ is the lifespan of the current individual, $n$ is the current year in its developmental process, $S_{N\text{max}}$ is the variable from the genetic algorithm which determines the upper limit of $S_N$, and $S_N$ is the minimum stress needed to grow an adjacent cell.
NEW POTENTIAL ELEMENTS

Next, the MD routine calls a subroutine, *NewElem*, that creates the array of potential new elements. The subroutine loops through all of the current elements and determines if it is necessary to grow a new element to alleviate some of the stress, induced from a load on the structure, on the current element. It is necessary to create a new element if the current element’s stress is greater than the $S_N$. If the stress of the current element is less than $S_N$, the routine moves on to the next element. If element $j$ is a candidate for a new element, it is checked for border edges. If it does not have a border edge, it is an internal element and cannot grow an element. If $j$ has either one or two border edges it will grow potential elements from those edges. Once a potential element is created a new node is added to the node matrix. This new node is the free node on the new element. The node is placed based on the area of $j$ and the location of the nodes on the growth edge of $j$. Figure 3-4 shows element $j$ that grows an element from each edge. Each new element has the same area as the original element, which is calculated in a subroutine:

$$\text{Area}_a = \text{Area}_b = \text{Area}_c = \text{Area}_j$$
Also, each new element has a different length edge to grow from and therefore a different overall shape (ie. element $b$ is wider than element $a$). The dashed arrows are the perpendicular bisector of the growth edge which leads to the location of the new node based on the calculated element area.

The mean of the two nodes on the growth edge is taken and a scaling factor is added to project the potential element's new node out perpendicular from the midpoint of the growth edge with an identical area as $j$. The new element then receives $j$'s properties such as the set of nodes that it is made of, the growth edge and the stress.

MERGING NEW NODES

Once the structure $GrowElem$ is created, the subroutine $MergeNewNodes$ is called. As mentioned in the above section, when a potential element is created, a new node is also created. Typically there are a variety of new nodes, and some may be very close to each other, or even at the same location. If the distance between two new nodes is found to be less than some predefined $EdgeMin$, the second element's free node is merged with the first element's free node using the $MergeNodes$ subroutine. In Figure 3-5, the new nodes $nn1$ and $nn2$ have a
distance, \( \text{dist} \), which is less than \( \text{EdgeMin} \). The nodes \( nn1 \) and \( nn2 \) merge at the midpoint of the separation and one of the new nodes, \( nn2 \), is eliminated.

MergeNodes performs three functions. First, it checks to see if any other element shares the node that will be eliminated. If so, it changes that element’s node to the newly merged node. Second, it decreases the node number in all of the elements of any node greater than the node to be eliminated because one node was eliminated. Finally, it eliminates the node that is a duplicate.

MERGING OLD NODES

Next, once the \textit{MergeNewNodes} subroutine is complete, the \textit{MergeOldNodes} subroutine is called. This is similar to \textit{MergeNewNodes} except it merges the nodes created by the development of new elements with the existing structure from the previous year. This subroutine loops through all of the remaining new nodes and compares them to the nodes that existed in the individual from the previous year. A node is merged with the closest node if it is less than some predefined \( \text{EdgeMin} \). If it is able to be merged, the node created by the new element is eliminated and the \textit{MergeNodes} subroutine is called, just as in \textit{MergeNewNodes}. The new node is always merged to the node from the
previously existing structure. In Figure 3-6, the new node \( nn1 \) has a \( dist \) which is less than \( EdgeMin \) with the node on \( Elem2 \). The node is merged to the previously existing structure and \( G1 \), the growth element, fills the gap left by \( Elem1 \) and \( Elem2 \).

Once all of the nodes have been merged within the specified \( EdgeMin \) distance, the structure can have some problems with element positioning. The subroutine \( MergeNewElems \) rectifies the two problems which can occur. First, if two elements share the same three nodes they are overlapping elements. In this case, one of the elements is removed from the \( GrowElem \) structure, and the other element continues to occupy that location. Figure 3-7 demonstrates how overlapping elements can occur and what happens to remedy this situation. The second problem that merging the nodes can cause is creating a degenerate element (an element consisting of a line and two nodes). In the case that an element becomes a line, the element is eliminated because it serves no purpose. This situation only occurs when new nodes are merged and then the free node is merged to the original structure. Figure 3-8 exhibits how a line element can be formed and how that situation is alleviated.
MAKING POTENTIAL ELEMENTS ACTUAL ELEMENTS

Once all of these subroutines are completed, the list of GrowElems, the potential elements, is added to the end of the list of existing elements one by one. As each one is added, it is also checked to see if it shares a common edge with another element. At least one edge will be shared with another element, the edge that it was grown from, but it can also have common edges with other elements from the merger of nodes. The MergeNewEdges subroutine is called and can do one of two things for the edges of the newly added element. First, if it is found to have a common edge with another element, the edge of the element in question is assigned to the existing edge and the Edge structure replaces the empty edge with the newly grown element. If the edge in question is not common with another edge from the existing structure, an edge is added to the Edge structure and is assigned to the newly added element. After the edges are assigned to the new
element, the nodes and material are assigned. The material is the same as the
parent element and the nodes are determined previously in the routine. The new
element is given an age of zero and a stress of zero. Finally, the area of the
element is determined and assigned by calling the subroutine \textit{ElemArea}.
\textit{ElemArea} simply determines the area of each element based on the location of
the three nodes given as:

\[
Area = \frac{1}{2} [(x_2 - x_3)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_3)]
\]

Once all of these assignments are complete for each of the remaining potential
elements in \textit{GrowElem}, the process of growing a new element is finished and the
program returns to the metamorphic development, MD, routine.

**EXPANSION OF EXISTING CELLS**

**GROW STRESS MAXIMUM VALUE**

Similar to \textit{NewElem.m}, \textit{GrowElem.m} is a routine that uses mesh
properties and inputs from the genetic algorithm as inputs. Also, similar to the
\(S_N\) determined in the \textit{NewElem.m} routine, a variety of different variables are set
at the beginning of the routine. The \(S_G\) variable for this routine is given as:

\[
S_G = \left[ \tanh \left( \frac{8n}{N} - 4 \right) + 1 \right] \left[ \frac{S_{G_{\text{max}}}}{2} \right]
\]

where \(S_{G_{\text{max}}}\) is the variable from the genetic algorithm which determines the
upper limit of \(i\)'s \(S_G\), and \(S_G\) is the minimum stress element \(j\) in individual \(i\) must
have to grow. The curve is similar in form to the one in \textit{NewElem.m}. The
variable \textit{factor} is defined by the user and determines what factor of the current
size of an element by which each node moves in each direction. If \textit{factor} is a low
number, the elements will only expand a minimal amount which will leave the structures shape relatively unaltered by element growth. If the factor is high, the structure will most likely change shape considerably. Next, the structure \textit{OldNode} is created using the matrix \textit{Node} to keep track of the position of all the nodes at the start of the growth process. The routine loops through all of the elements, and if \( j \)'s stress is greater than that of \( S_G \) it can be grown.

**DETERMINING THE NODAL LOCATIONS**

Next, the element's centroid is determined by taking the average of its nodal coordinates. Because the loads and constraints cannot be moved, the routine loops through to see if a node on the element in question is coincident with a load or a node. If so, it will not be displaced. If it is not under that constraint, a \( dx \) and \( dy \) are determined. In the case of \( dx \), the difference between the \( x \) location of the current node and the element's centroid's \( x \) location are multiplied by the growth factor to change the \( x \) location of the node. The same applies to the \( dy \), but in the \( y \) coordinates.

If the current node has already been grown in some direction, the new displacement is added to the previous displacement of that node. This will ultimately end in a vector summation and canceling effect from directly opposing directional and equal magnitude growths on the same node. Figure 3-9 shows this vector summation of the \( y \) directional displacements and the vector cancellation of the \( x \) directional displacements. If it is a node that has not yet been displaced, the structure \textit{Disp} receives a new addition. Finally, once all of the elements in an individual structure have been evaluated, the routine
CSTSolver.m is called to determine the new Elem structure. By calling CSTSolver.m, the nodal locations are determined. In the routine CSTSolver.m, using finite element modeling techniques, the displacement of the nodes can be determined based upon the nodal coordinates and loads from the GrowElem.m routine. This subroutine uses forced displacements to determine the new nodal locations of the elements. Thus the growth process is completed and returns to the routine MD.

One major hindrance to the successful running of the entire program is if the variable factor is not large enough the bridge will not grow. In light of this situation, factor was changed to a design variable, allowing the program to select a variety of different factors for the different individuals.

ELIMINATION OF UNNECESSARY CELLS
VARIABLE FOR THE REMOVAL OF ELEMENTS

The function KillElem.m, which eliminates the elements that are considered unnecessary because they carry very little to no stress, uses a variety of different subroutines and checks to eliminate each unnecessary element while keeping the structural integrity of the individual. As in NewElem.m and
GrowElem.m, the KillElem.m function begins by setting the variables, of which $S_R$ is the removal stress, or the stress below which an element is considered superfluous. $S_R$ is given as:

$$S_R = \left[ \tanh\left( \frac{8n}{N} \right) - 4 \right] + 1 \left\lceil \frac{S_{R_{\max}}}{2} \right\rceil$$

where $S_{R_{\max}}$ is the variable from the genetic algorithm which determines the upper limit of $i$'s $S_R$, and $\text{RemStress}$ is the minimum stress element $j$ in individual $i$ must have to stay intact. In contrast with $S_N$ and $S_G$, the $S_R$ curve is tested to see if an element is beneath the curve rather than above it.

POTENTIAL ELEMENTS TO BE REMOVED

The first subroutine called is the MakeRemElem. This subroutine does just as the function name implies and makes the structure RemElem, a matrix of all viable candidates for extermination. First, a matrix of all the loads and constraints is created. This is used to avoid the elimination of any elements that have a node in common with a constraint or a load. The subroutine then proceeds to loop through all of the elements in stress order, beginning with the elements with the least stress on them and proceeding on working to the elements with the most stress on them. There are three conditions that an element must meet to be eliminated. First, it cannot have a node with a load or constraint. Eliminating elements that have low stresses attached to constraints or loads will be seen to in future research. Currently, it was decided that the programming was more complex than the just value of eliminating these elements. Secondly, it must have a stress of less than the current $S_R$. Thirdly, the
element must be at least one generation old. This constraint is employed to allow new elements the ability to live for one generation even if they are not under much stress the first year of their lives. They may become valuable once another generation of elements grows on the periphery. Also, this prevents bridge dwarfism, the continuous slaughter of new elements stunting the growth of the entire individual.

CHECK FOR BORDER EDGES

Next, the routine loops through each element that has become a candidate for removal. The first check of an element is to see how many border edges it has. The subroutine \textit{BorderEdge} is called and simply returns a zero if the element is an interior element, a one if it has one border edge, or a two if it has two border edges. If it returns a three or greater something has been computed incorrectly because it should not be floating nor should it have more than three edges.

CHECKING THE MOBILITY OF THE STRUCTURE

To ensure that the structure is still rigid, the routine, with the \textit{BorderEdge} information, enters into a switch, a conditional loop in MatLab, with three possibilities: one, two or zero for the length of \textit{BorderEdge}. In any of the cases, the subroutine \textit{CheckMobility} is called, which checks to see if the individual will become a mechanism if the element in question is eliminated. The mobility check begins by setting the strain-displacement matrix. Next, looping through all of the elements, an elemental strain-displacement matrix is created and added to the global strain-displacement matrix. Once this is done, the contributions from the constraints are added to the global strain-displacement matrix. Finally, the
mobility based upon rank of strain-displacement matrix is calculated, appropriately given the variable name $DOF$. If $DOF$ is zero or less, the element can be eliminated.

REMOVAL OF THE INTERIOR ELEMENTS

In the case of an interior element, two processes occur. First, the element is eliminated from the $Elem$ structure. Secondly, the subroutine $DecEdgeElem$ is called. This subroutine decrements the element number in the Edge structure if the element is greater than the number of the freshly eliminated elements and creates a border edge where the freshly eliminated element leaves borders on other elements. In Figure 3-10, elements 1-9 and 13 are internal elements.

REMOVAL OF AN ELEMENT WITH ONE BORDER EDGE

In the case of an element with one border edge, the process is very similar to an interior element with one exception. Because the border edge is unrelated to any other element in the individual, the edge must be eliminated along with the element being eliminated. If the edge is not eliminated, the structure Edge will have extra, unnecessary information. Once the element is eliminated and $DecEdgeElem$ is called, the edge which is a border edge is eliminated and $DecElemEdge$ is called. This subroutine reduces the edge numbers in the structure $Elem$ by one where the edge is greater than the one which has been deleted. In Figure 3-10, elements 10, 14, 16 and 17 have one border edge.

REMOVAL OF AN ELEMENT WITH TWO BORDER EDGES
In the case of an element with two border edges, the process is very similar to an element with one border edge with one exception. Because the two border edges can form a node that is unrelated to any other element, that node needs to be eliminated. If the node is shared with other elements, it is left alone. First, the node is eliminated. Next, the subroutine \textit{DecEdgeNode} is called, which decreases the node number in the \textit{Edge} structure for all of the nodes that are greater than the one which has been deleted. After this, the subroutine \textit{DecElemNode} is called, which in like fashion decreases the node number in the structure \textit{Elem} for all nodes that are greater than the node which has been eliminated. The routine proceeds to call \textit{DecEdgeElem}, described above, and \textit{DecElemEdge}, also described above. The subroutine \textit{DecElemEdge} is called twice, once for each edge which is being eliminated from the individual. The element is then eliminated and the routine moves on to the next element in the \textit{RemElem} matrix.

In Figure 3-10, elements 11, 12, 15 and 18 have two border edges with free nodes.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3-10.png}
\caption{Types of Elements for \textit{KillElem.m} (\#17 has One Edge; \#18 has Two Edges; \#7 is Interior)}
\end{figure}
CHAPTER 4: OPTIMIZATION VIA THE GENETIC ALGORITHM

OPTIMIZATION TECHNIQUES IN GENERAL

As engineers, we strive to make processes work efficiently and arrive at the best answer to the problem at hand. This is often accomplished by applying an optimization technique. Many optimization techniques exist for a variety of different problems and circumstances, many of which are discussed in Optimization Concepts and Applications in Engineering co-authored by Tirupathi R. Chandrupatla and Ashok D. Belegundu [2]. Some examples given in Matlab's optimization toolbox include unconstrained nonlinear minimization, quadratic and linear programming, and constrained linear least squares, to name a few. Each form of optimization should reach the same optimum solution for a given problem, although different methods have different efficiencies and chances of successfully attaining that optimal solution. For this project, it was decided to use a genetic algorithm to optimize the variables.

ADVANTAGES OF THE GENETIC ALGORITHM FOR OUR PURPOSES

The beauty of the genetic algorithm lies in its ability to converge upon an answer while remaining random enough that it does not fall into a local minimum and is likely to find the global optimum in several tries. In most gradient-based optimization techniques, once all of the answers are converging into one answer, there is no way to find another possible solution, while in the genetic algorithm the ability for weak members to carry characteristics to the next generation gives it the ability to remain quasi-random and search a larger
design space. The basic genetic algorithm is based upon the concept of population optimization, also known as survival of the fittest. Because we are dealing with only one species, namely finite element structures, there is no interference of stray variables from other developing structures. For example, if the constraints and loads were being optimized simultaneously, the overall optimization would not work properly. There are no external variables to the survival of a bridge other than its own genetic information relative to the other bridges grown in the same generation.

**VARIABLE AND CONSTANT DEFINITION**

As in many engineering applications, the users must enter a set of information to tailor the process to his needs. In the genetic algorithm there are two types of user inputs, namely variables that are being optimized and constants. For our purposes, the constants defined by the user are:

1. **StressMax** – The maximum value of stress an element can withstand without a penalty
2. **pop** – The number of members in the population
3. **lenGene** – The number of genes which define each variable characteristic
4. **percent_keep** – The percentage of members to keep from one generation to the next
5. **nXover** – The number of places the parent information is crossed over with its mate
6. **deviation** – The percentage of structures that is acceptable to be different than the rest of the population at which the program has found as the best solution
7. **mutt** – The chance for a member of the population to have a genetic mutation
The variables that will be optimized are:

1. \( r_{\text{DeathYear}} \) – The number of years a bridge will develop
2. \( r_{\text{NewStressMax}} \) – The stress level required for an element to create another element
3. \( r_{\text{GrowStressMax}} \) – The stress level required for an element to grow in size
4. \( r_{\text{KillStressMax}} \) – The stress level required for an element to be eliminated

THE INITIAL POPULATION

Once these variables and constants have been defined by the user, the program is ready to start working. One interesting fact about the genetic algorithm is the constant use of the random number generator to create the initial population’s binary genetic information. This randomness allows the program freedom to explore possible answers that the user may not consider anywhere close to the optimum. This method results in a lot of poor results in the beginning, but as mentioned before it also avoids getting stuck in local minima. The program GA calls the subroutine Genesis.m, the initial population generator. This subroutine fills the population, \( p \), with genetic information. Each gene in the structure \( p \) is represented by either a zero or a one. Figure 4-1 gives an example of one initial population with five individuals, each having four genes for each of the four variables.
CONVERTING FROM BINARY GENES TO REAL WORLD INFORMATION

Once the initial population has been given genetic information, the setup process is complete and the optimization can begin. Because the genetic information is in binary form, it must be converted to actual variable numbers. Calling the subroutine Convert_Genes.m, each of the individuals in the population have their binary genetic information changed into variables that lie within the ranges given by the user, as mentioned in the above section. Using the interpolation:

\[
Variable = r_{High} - [r_{High} - r_{Low}] \left[ \frac{b_{High} - b_{Variable}}{b_{High}} \right]
\]

where \(Variable\) is the final number that will be used in the genetic algorithm for the individual and variable in question, \(r_{High}\) is the range maximum, \(r_{Low}\) is the range minimum, both user defined, \(b_{High}\) is the highest number the binary set can attain and \(b_{Variable}\) is the value of the binary set being converted. All the resulting variables are rounded to the nearest integer. The following equation shows how individual \(p(1)\)'s \(DeathYear\) from Figure 4-1 would be converted if the...
range went from five to fifty. Because each variable has four genes, the \( bHigh \) is fifteen. The \( bVariable \) is eight because the binary representation is \( 2^3 \).

\[
29 = 50 - [50 - 5]*\left[\frac{15 - 8}{15}\right]
\]

For the first individual, the \( DeathYear \) would be 29, given the range above.

The genetic information that has been converted into useable numbers is now entered into the metamorphic development process.

**CHECKING FOR CONVERGENCE**

Once all of the structures in one generation have matured, the genetic information of each structure in that generation is checked to see if convergence has occurred. The subroutine \( \text{Did\_It\_Converge.m} \) checks to see if the population consists of somewhat identical individuals. Essentially, the subroutine finds the average deviation away from the mean value of each variable. Once the four deviations are calculated, the average of the four deviations is given back to the main program as \( \text{converge} \). If the value of \( \text{converge} \) is less than \( \text{deviation} \), the population has converged to the precision predefined by the user and the program is complete. If not, the program continues to the next generation. All the routines and subroutines described throughout the rest of the chapter only occur if the program has not converged. If the program has converged, we have successfully come to our optimal solution.

**GOAL OF OPTIMIZATION FOR THIS THESIS**

With any optimization technique some property or variable needs to be the object of optimization. In our case, we are trying to minimize the total mass of a
structure while maintaining its structural integrity. The structural integrity as defined in this thesis is the ability for each element to have stresses less than the yield stress of a given material. How the mass is determined will be discussed in the section dedicated to the integration of the metamorphic development and the genetic algorithm. The genetic algorithm receives a mass, \textit{p\_mass}, for each individual in a generation. This mass determines how fit an individual is in comparison to the rest of the population. The \textit{p\_fitness} of each individual is calculated as the difference between the heaviest individual and the individual in question. The \textit{p\_fitness} is used in the subroutine Roulette.m.

\textbf{DEVELOPING THE NEXT GENERATION OF BRIDGES}

\textbf{THE SURVIVORS FROM THE PREVIOUS GENERATION}

At this point we begin to develop the \textit{child} generation. The first step in creating the \textit{child} generation is keeping a few candidates from the current generation. In the constants section above the constant \textit{percent\_keep} was defined as the percentage of members to keep from one generation to the next. Using both \textit{percent\_keep} and \textit{pop}, we can create the variable \textit{keepers}, the actual number of individuals to keep from one generation to the next. The program keeps some individuals from one generation to the next at random, but with a weight towards the more fit individuals. Calling the subroutine \textit{Layover.m}, for the number of \textit{keepers} a random individual is selected to live on using the \textit{Roulette.m} subroutine.

\textbf{THE ROULETTE METHOD}
In *Roulette.m*, the underlying concept is that every individual in the population has a chance to be selected for survival, but the fittest individuals will have a greater opportunity to be selected than those less fit. First the roulette wheel is created using the *p.fitness*. Figures 4-2 and 4-3 show two visually different, yet conceptually identical analogies for the roulette concept. In Figure 8, the shape of a roulette wheel is used with the greatest space given to the most fit individual and less space for each subsequent individual based on its fitness.

![Figure 4-2: Roulette Wheel](image)

Figure 4-2: Roulette Wheel

Figure 4-3 is more true to the way the code is written, which looks like a linear strip dart board. Each part of the board can be hit, but with a random throw it is more likely to hit the larger area spaces. Once *keeper* individuals are selected via the roulette method, the *child* structure is filled with the genetic information from the individuals that were selected to layover to the next generation. This is calculated through a random number generator.

**CROSSOVER OF PARENT BRIDGES**

Next, the subroutine Crossover.m is called. This subroutine fills the remainder of the *child* structure with hybrids of the individuals in the current generation. In most typical crossover methods there is only one place for the
crossover to occur, but in this program the user has the ability to choose the number of places crossover occurs. The benefit of increasing the places of crossover is randomness, but if there are too many places for crossover the answer will probably never converge. First, two parents are selected using the Roulette.m subroutine mentioned above. Once two distinct individuals have been selected the subroutine takes genes from both parents switching at crossover points. The first parent to contribute genes to the offspring is always the first parent selected. This is displayed in Figure 4-4, where two parents are selected with two crossover points. The offspring of this mate can be seen at the bottom of Figure 4-4. Between two and four crossover points were used in this thesis.

MUTATION OF BRIDGES IN THE NEXT GENERATION

At this point, the entire child structure is filled with either individuals from the previous generation or created individuals from the crossover technique. If this remained the case, the genetic algorithm would not properly imitate real genetic passing of genes and therefore we call upon the subroutine Mutation.m. In this subroutine, each gene of each individual has some chance to change from the binary one to zero and vice versa. Defined in the main routine, the variable mutt is the percent chance that a gene will undergo a genetic mutation. The
subroutine marches through each gene and at each gene a random number between zero and one is generated. If the random number is less than the decimal representation of mutt the gene will undergo mutation. With this in mind, the variable mutt will usually be between 1% and 0.01%, which was used when running the program for the purposes of this thesis. If the percent of mutation is too high, the population will never converge because the children will not represent the parents well, but will be mutants.

The final step in the genetic algorithm is the transfer of identity from child to parent. Once the child structure is filled and mutated, the structure $p$ is replaced by the child structure and the process continues from the beginning. The new $p$ structure is grown and evaluated.
CHAPTER 5: OPTIMIZATION OF THE METAMORPHIC DEVELOPMENT PROCESS VIA THE GENETIC ALGORITHM

This section is the focal point of this thesis. In the previous pages both the metamorphic development process and the optimizing genetic algorithm have been thoroughly explained. Here, the synthesis of the two will be explained in detail.

THE GENERIC GENETIC ALGORITHM

The genetic algorithm was described before specifically applicable to this thesis, but in general the genetic algorithm is used to optimize an equation, just like other optimization techniques. Other optimization techniques are typically faster than the genetic algorithm when finding the solution to a gradient based, continuous, linear problem because there is not as much ability for variance in the optimized solution as the program is executed. This is not the case in our situation, though, and therefore the genetic algorithm is an appropriate choice. Our problem is highly nonlinear, discontinuous and therefore using a gradient based approach would make little sense. The variation of one variable could change the solution an undetermined amount. Also, because we would like to mimic the development of a human population, the genetic algorithm is an appropriate choice.

THE NEED FOR A GENETIC ALGORITHM

The metamorphic development process is just that, a process. The structure resulting from the variables taken from the genetic algorithm and used
in the metamorphic development process cannot be predicted by the mind of the user unless the program has the opportunity to run through the organic process of growth, development and decay. This is very similar to the development of human physiology. Although genetic scientists can see the activated genes in the DNA, no scientist can determine what a person will look like fifty years into their life from the genetic data alone. This is the uniqueness of this integration of the genetic algorithm with the metamorphic development process. As the genetic algorithm converges to an answer it is not looking towards a specific set of variables that fill placeholders, but rather a set of variable conditions that will develop the structure into an ideal structure for the application at hand.

FLOW OF CONTROL IN THE GENETIC ALGORITHM

As Figure 5-1 shows, the flow of control is not the same as a typical genetic algorithm when analyzing the optimization parameters. In a typical genetic algorithm, the input would be some variables, the equation would return the
solution and the answer would be the optimized parameter. If the user was minimizing or maximizing the answer, the solution would tend toward either extreme. If the user wanted to hit some target number, the answer would be evaluated in comparison to that number. In this case, though, the answer has two parameters, namely the mass and a penalty added to the mass if the structure is too highly stressed, but both are expressed in one number.

**EVALUATION OF EACH BRIDGE**

At the conclusion of each structure's development, the subroutine `GetMass.m` is called. This subroutine returns the mass of the structure. For each element that is under the critical stress `StressMax`, the actual mass of the element is added to the total mass of the structure. For all of the elements that have a stress greater than `StressMax`, the actual mass of the element is calculated and multiplied by a *penalty* factor and the product is added to the total mass. Using
this method solves two problems in one stroke. In these problems, the absolute optimum is to have the least amount of material being used while still meeting all of the requirements set forth by the user with regards to the level of stress in the entire structure. Therefore, the mass of each element is the objective function. Adding a substantial amount of mass for every element that does not meet this requirement forces the program to have a higher likelihood of choosing structures that have no weak members. The mass of each element becomes the primary fitness factor in the genetic algorithm as mentioned in previous chapters.
CHAPTER 6: RESULTS AND CONCLUSIONS

GOAL OF THE THESIS

The overall goal of this thesis is to develop a program that seeks to increase the stress to mass ratio of a structure in comparison with a flat bar while not exceeding the yield stress.

EFFECT OF THE GROWTH FACTOR

As mentioned in Chapter 3, the ability for elements to grow in size circumvents the need for a grid based mesh. In order to ensure that the routine GrowElem.m was working properly, a test was executed with just the GrowElem.m as the acting routine on the bridge. This resulted in a surprising result. For the simply supported situation, the constrained nodes and the loaded node were fixed in the position at which it started during the periods when the elements could either grow or add another element. All of the other nodes on the bridge were able to move freely as needed. As the bridge developed, the elements that were in between a constraint node and the load began to arch upwards. This resulted finally in the general shape of the bridge becoming like an $M$, as seen in Figure 6-1. This trait can be seen in all of the bridge developments that included a significant factor of growth to the elements. This was an interesting effect because it shows that in some cases the Michell structure is not necessary to meet the design constraints even though a straight bar will not suffice. The $M$ shaped structure is a low material solution to some cases in which the Michell truss may be overkill and the straight bar would fail. The Michell structure may be overkill because the element would become so thin that the actual solution, although
mathematically feasible, would be difficult or impossible to create. The straight bar would fail using an equivalent amount of material as the \( M \) shaped structure, and therefore for some cases the \( M \) shaped structure would be an optimum solution.

**CONVERGENCE**

In a twenty-generation execution with a population size of fifteen, the general population converged over five variables. Table 6.1 below shows the first generation and the data that was randomly chosen with the resulting masses acquired from those random inputs. Table 6-2 below shows the twentieth generation of development with the resulting masses. As seen in Table 6-1, a majority of the variable data is unrelated to the other data from the other variables, whereas in Table 6.2 the data is clearly similar in any variable set. The masses of the first four structures are quite different from those of the other structures; this arose from crossovers incurred from generation nineteen that deviated from the converged upon result.

<table>
<thead>
<tr>
<th>Bridge</th>
<th>DeathYear</th>
<th>NewStress</th>
<th>GrowStress</th>
<th>Factor</th>
<th>KillStress</th>
<th>Mass Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>163729</td>
<td>758979</td>
<td>4080559</td>
<td>45311</td>
<td>5.403784e-001</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>913110</td>
<td>624519</td>
<td>1803751</td>
<td>339427</td>
<td>3.497628e-005</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>914808</td>
<td>145789</td>
<td>29878970</td>
<td>325129</td>
<td>2.504976e-006</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>540060</td>
<td>690280</td>
<td>1930570</td>
<td>637814</td>
<td>3.219903e-001</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>273280</td>
<td>904642</td>
<td>15013762</td>
<td>216591</td>
<td>2.266259e-006</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>626241</td>
<td>724594</td>
<td>47869857</td>
<td>849871</td>
<td>1.399867e-006</td>
</tr>
<tr>
<td>7</td>
<td>21</td>
<td>518786</td>
<td>799027</td>
<td>4498141</td>
<td>169958</td>
<td>3.896950e-001</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>52600</td>
<td>444916</td>
<td>907432</td>
<td>72797</td>
<td>2.426997e-006</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>427583</td>
<td>439416</td>
<td>4235144</td>
<td>92989</td>
<td>2.419924e-006</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>458994</td>
<td>46314</td>
<td>1035039</td>
<td>497235</td>
<td>3.299164e-006</td>
</tr>
<tr>
<td>11</td>
<td>17</td>
<td>46092</td>
<td>556914</td>
<td>27169980</td>
<td>20381</td>
<td>2.075428e-005</td>
</tr>
<tr>
<td>12</td>
<td>18</td>
<td>917702</td>
<td>26306</td>
<td>48861643</td>
<td>587775</td>
<td>3.193300e-001</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>710323</td>
<td>307757</td>
<td>25196727</td>
<td>612099</td>
<td>7.992430e-006</td>
</tr>
<tr>
<td>14</td>
<td>9</td>
<td>642826</td>
<td>808200</td>
<td>25281779</td>
<td>61412</td>
<td>2.854678e-006</td>
</tr>
<tr>
<td>15</td>
<td>22</td>
<td>195441</td>
<td>855846</td>
<td>27441825</td>
<td>698740</td>
<td>4.456850e-001</td>
</tr>
</tbody>
</table>

Table 6-1: Generation One Data
(Randomized Variables)
PRELIMINARY STRUCTURE IMAGES

In a similar fashion to human beings, when different variables are dominant in the genetic makeup of a structure the resultant structure displays those traits. The following examples display graphically what happens to different structures when different genes are dominant.

There were a variety of different results from the randomized data that created the first generation of bridges. Each of the following depictions will display a different variable that was accentuated. In Figure 6-2, structure one from generation one is shown. The three features that are expressed clearly are the factor, the GrowStress and NewStress. The high factor and GrowStress

![Figure 6-2: High Factor, GrowStress and Low NewStress]
in the curviness of the structure while the high \textit{NewStress} resulted in an array of new elements on the structure.

In Figure 6-3, the tenth bridge in the first generation, expresses a low \textit{factor} and a high \textit{KillStress}. This can be seen in the fact that there is little to no bending in the actual structure, yet there is an elimination of unnecessary elements. The low \textit{factor} results in a relatively straight structure while the high \textit{KillStress} shows in the elimination of a variety of elements.

\textbf{THE CONVERGED ANSWER}

The population converged upon a solution that has the proper mix of all five of the variables, namely the \textit{DeathYear}, \textit{factor}, \textit{GrowStress}, \textit{NewStress} and \textit{KillStress}. The graphic in Figure 6-4 clearly shows that the bridge survives at a \textit{MaxStress} of 15,000, yet is at a lower mass than Figure 6-2, which also passes the stress test. As seen in Figure 6-4, the bridge has a stress level close to 15,000, but does not exceed it. This gives a solution that is highly stressed, but will not fail,
therefore we can logically conclude that it is either the best solution or close to the best solution to the given situation because the material is being entirely utilized. Looking at the statistics of this structure and comparing it to the original beam that it developed from, we can see a definitive improvement of stress versus mass. In the original, there was a total mass of 28,080 kg and an average vonMises stress (throughout the entire structure) of 27,000 N. Also, as mentioned above, the maximum allowable stress was 15,000 N and therefore most of these elements would fail in any case. The average vonMises stress per mass is 0.962. The new structure weighs in at 64,350 kg, approximately 2.3 times the weight of the original, but has an average vonMises stress of 6,500 N throughout the structure. All of the elements are feasible and the average vonMises stress per mass is 0.101, a dramatic improvement.
FINAL CONCLUSIONS ABOUT SIMPLY SUPPORTED STRUCTURE

The above figures provide a visual understanding of the inner workings of the program. The first generation yielded some randomly generated bridges of which some were feasible and some were not. All of the feasible bridges had a mass that was higher than that of the final solution in generation twenty. The results from the convergence of the variable data give us a solution that is not only feasible, but rather is the most highly stressed feasible solution, which naturally has the least amount of mass.

WHY DID THE BRIDGE BECOME THIS SHAPE?

The pressing question now is why did the structure form into the shape that it became? For the purposes of this work the phenomenon in which the bridge morphs from a straight structure into a double bowed structure will be called “M-ing”, because the bridge begins to look like the letter M. There are a few conjectures that can be made regarding this M-ing phenomenon, based on some failed attempts at running the program while it was being developed.

FAILURE RUN WITHOUT CONSTRAINED LOADS

In one trial run, while the code for keeping the nodes with point loads stationary was not completed, an interesting sequence of growth occurred. During this run, NewElem.m was turned off and therefore the elements were not allowed to duplicate in order to relieve some expensive computing power. Because the point load was able to freely float, but the constraints were set at
fixed locations, the program tried to accommodate those conditions and make the best structure. In doing so, the elements all changed with regard to their dimensions, but did not duplicate as mentioned above. All of the elements elongated and began to push the point load upwards, which created an arch that developed an increasingly rounded bend though the generational growths. When the growth acquired a structure that met the conditions set by the material properties as well as having the least amount of mass possible it was an arch. As seen in Figure 6-7, the arch structure allows for the compression lines to travel from the loaded node, through the structure, to the points of the constrained nodes. As we recall from solid mechanics, most materials have a higher capacity to withstand compressive or tensile forces rather than bending forces. By thrusting the structure upwards from a linear platform with the load in the center to an arch with the load at the crest, the bridge’s growth tried to displace the bending loads it experienced with compressive loads. To make reference to the amount of improvement that the arch had over the straight bar, the masses and maximum stresses at both states were taken. The mass of the straight bar was 6 kg and the vonMises stress upon it was 95,333 N. The arch weighed in at 9.2627
kg, just over 150% of the original, but the vonMises stress was impressively reduced to 20,513 N, less than \( \frac{1}{4} \) of the original stress.

![Figure 6-6: Straight Bar and M-ing Model Stresses](image)

**COMPARISON OF THE BAR AND M-ING**

In order to verify the structural strength of the M-ing topology, a pair of models was created in SolidWorks. A model of a simply supported straight bar was created with a mass of 0.72 lbs. A second model with the M-ing topology was created with the same mass and supported at the same points. The maximum vonMises stress endured by the straight bar was 9.904 psi while the maximum vonMises stress endured by the M-ing model was 8.016 psi. These models and their maximum vonMises stresses are seen in Figure 6-6. This verifies the structural superiority of this topology in comparison.

**EXPLANATION OF THE M-ING EFFECT**

In light of the failure run that was just mentioned, when the code was completed properly for maintaining the position of a loaded node, another
interesting phenomenon occurred. Again, in an attempt to conserve some computing power, the bridge was only capable of expanding its elements and not duplicating. As before, the bridge attempted to alleviate the bending stresses that it was incurring by arching from the constrained nodes to the load node in the center, but the central point was unable to move upwards to create an arch-type bridge. In its attempt to arch up, the fixed nodes at the constraints and the fixed node in the center which was applying the load caused this M-ing phenomenon to occur, which can be seen in Figure 6-8 as a silhouette with a centerline through it. By M-ing, the bridge lightened the bending load on the constrained nodes by transforming this bending load to both a tensile and compressive load. The tensile forces, acting from the outside of the three arcs in the M, and the counterpart compressive forces, acting on the inside of the three arches, replace the bending forces at the constraints. If the bridge does not change its shape to transform the forces acting on the constraints from bending forces to tensile and compressive forces, the bridge would require massive supports at the constraint nodes. This suggestion would necessarily solve the problem at hand, but would not be an efficient solution.

![Compression lines](image)

Figure 6-7: Archway Compression Lines
ARRIVING AT A SEMI-MICHELL STRUCTURE

Now, after having understood the $M$-ing phenomenon, it is sensible that we further explain why we arrived at the model that is our solution in this case. The aforementioned test run which fleshed out the $M$-ing effect was done with a minimum load. Our primary goal during that test was to see if the loaded node would maintain its position, not to find a ‘good solution’ to a real world problem. In any case, the real question is how we relate this $M$-ing effect to the semi-Michell structure that was our resultant bridge, or even the actual Michell structure which is commonly accepted as the ideal solution to these problems in the field of topology optimization.

With the real world constraints bearing on the bridge, a different situation exists than the previous scenario. At this point our elements can duplicate when it is necessary, the different genetic information is taking effect, and the real world material properties are being used. At the point that we arrive at an optimal solution, namely the last generation, we can see that the structure looks almost like the Michell structure with three primary differences. First, the structure is generally flatter than the Michell structure. This may be a result of
the general constraints such as the amount of loaded force, the span of the bridge, the height of the bridge or the amount that the structure was fragmented into finite elements. The other significant difference between our result and that of Michell is the cap of the structure. Our structure is lacking the rounded cap that the general Michell structure has. This can also be attributed to a variety of factors such as the magnitude of the applied force on the bridge, the span of the bridge, the initial height of the structure or the amount that the structure was fragmented into finite elements. Finally, our bridge has much more material in the center than the Michell structure does. This is probably because in the particular situation at hand the environment required a thicker body than the typical Michell structure and did not allow more elements to be removed.

FUTURE WORK
The research of this thesis can and will be continued in the future. One area of interest for future work is the use of a small number of generations in order to eliminate the degeneration of elements. The program used for this thesis can run more efficiently with cleaner programming techniques which can be employed for future use.

REVIEW OF THIS THESIS' WORK
This thesis pursued two goals. Primarily, this thesis integrated the metamorphic development process with the genetic algorithm to optimize the design variables of the metamorphic development process. The results of this goal, covered in Chapter 6, show that the integration of the metamorphic development process and the genetic algorithm is possible and the results are feasible. The second goal of this thesis was the ability for the metamorphic
development process to work without the use of fixed nodal locations. This was accomplished using an array of numerically unrestrained coordinates for nodal locations rather than a predetermined mesh.
REFERENCES


Pergamon. PII S0045-7949(01)00137-7, 2527-2539.

APPENDIX A

MATLAB CODE
% This is the main program for the genetic algorithm process

close all
clear all
clc

clear all

dc

% Variables for the GA
rDeathYear = [2,45]; % USER DEFINED (variable(1))
rNewStressMax = [1,2000000]; % USER DEFINED (variable(2))
rGrowStressMax = [1,2000000]; % USER DEFINED (variable(3))
rFactor = [1,100000000]; % USER DEFINED (variable(4))
rKillStressMax = [1,2000000]; % USER DEFINED (variable(5))
range = [rDeathYear;rNewStressMax;rGrowStressMax;rFactor;rKillStressMax];
rLen = length(range);

% Constants for the GA
StressMax = 15000; % USER DEFINED
pop = 15; % USER DEFINED
lenGene = 24; % USER DEFINED
nGene = rLen*lenGene;
chunks = rLen;
percent_keep = 10; % USER DEFINED
keepers = round(pop*(percent_keep/100));
nXover = 1; % USER DEFINED
deviation = 2; % USER DEFINED
mutt = 2; % USER DEFINED

% Generate random first population
p = Genesis(pop,nGene);
converge = deviation + 1;
generation = 1;

for generation = 1:20
    p = Convert_Genes(pop,nGene,lenGene,p,chunks,range); % Converts genes in population to base_10

    for k = 1:pop
        LifeSpan = p(k).variable(1); % Output to screen the lifespan of the current bridge
        clear LifeSpan
        p(k).mass = MD(p(k).variable(1),p(k).variable(2),p(k).variable(3),p(k).variable(4), ...
p(k).variable(5),generation,k,StressMax); % Calls the metamorphic development program
    end
    converge = Did_It_Converge(p,pop,rLen);
    WriteGeneticData(p,generation,pop);
    p = Objective(p,pop); % Assigns a fitness to each individual
    child = Layover(p,pop,keepers);
    child = Crossover(p,nGene,nXover,pop,keepers,child);
end
% Creates a new member in the population
child = Mutation(child,pop,nGene,mutt);
p = child;
generation = generation + 1;
end

fprintf(1,'%s
','The population has converged.');

MD.m

% function mass =
MD(DeathYear,NewStressMax,GrowStressMax,factor,KillStressMax,trackgen,trackbridge,StressMax)
close all
clear all
clc
filename = 'twenty_mitchell.txt';
[nMaxNode,nMaxElem,Node,Elem,Con,Sym,Load,Prop] = FileIO(filename);
% Read in data from file
[Elem,Edge] = EdgeDetect(Elem);
% Detect edges of structure
Elem = CSTSolver(Con,Node,Elem,Load,Prop);
% Solve for stresses in truss
for z = 1:DeathYear % Each individuals dying year
    % Make all the elements one year older
    for i = 1:length(Elem)
        Elem(i).age = Elem(i).age + 1;
    end
    % Creates new elements when the stress is too high
    [Node,Elem,Edge] = NewElem(Node,Elem,Edge,Con,Load,Prop,DeathYear,z,NewStressMax);
    Elem = CSTSolver(Con,Node,Elem,Load,Prop);
    % Grows elements that have a moderate amount of stress
    Node = GrowElem(Con,Node,Elem,Load,Prop,DeathYear,z,GrowStressMax,factor);
    Elem = CSTSolver(Con,Node,Elem,Load,Prop);
    % Removes elements that are not bearing much stress
    [Node,Elem,Edge,Load,Con] = KillElem(Node,Elem,Edge,Con,Load,Prop,DeathYear,KillStressMax,z);
    Elem = CSTSolver(Con,Node,Elem,Load,Prop);

    mass(z) = GetMass(Elem,Prop,StressMax);
    % Function that returns the total mass of the structure
    tracker = WriteBridge(Node,Elem,Edge,Con,Load,trackgen,trackbridge,z)
    % Function that writes the bridge info to a file for future use
end
mass = GetMass(Elem,Prop,StressMax);
% Function that returns the total mass of the structure
function FileIO(filename)

i = 0;     % index variable
j = 0;     % index variable
k = 0;     % index variable

ITemp = zeros(5,1);  % temporary integer variables
fTemp = zeros(3,1);  % temporary floating point variable

nDim = 2;     % number of dimensions in problem

fid = 0;

[fid,message] = fopen(filename,'r');
if fid == -1
    disp(message)
    return
else
    fprintf(1,' %s%s
','The file ',filename,' was opened successfully')
end

% Begin reading data from input file
% %

header = fgets(fid);  % Skip over header lines

% Read in number of nodes and elements
% %

header = fgets(fid);  % Skip over header lines
header = fgets(fid);  % Skip over header lines
header = fgets(fid);  % Skip over header lines

nNode = fscanf(fid, '%d',1);  % Number of nodes
nElem = fscanf(fid, '%d',1);  % Number of elements
nMaxNode = fscanf(fid, '%d',1);  % Maximum allowable number of nodes
nMaxElem = fscanf(fid, '%d
',1);  % Maximum allowable number of elements

nDOF = nDim * nNode;  % Number of degrees of freedom
% Read in number of materials, constraints and loads

header = fgets(fid); % Skip over header lines
header = fgets(fid); % Skip over header lines

nMat = fscanf(fid, '%d', 1); % Number of materials
nCon = fscanf(fid, '%d', 1); % Number of constrained DOF's
nMaxCon = fscanf(fid, '%d', 1); % Maximum number of constraints
nLoad = fscanf(fid, '%d', 1); % Number of applied loads
nSym = fscanf(fid, '%d
', 1); % Number of symmetry lines

% Read in screen display option

header = fgets(fid); % Skip over header lines
header = fgets(fid); % Skip over header lines

iQuiet = fscanf(fid, '%d
', 1);

if (iQuiet == 0)
    fprintf(1, '%s %4d %s %6.2f
', 'Node number:', j, 'x-Coordinate:', Node(j,1), 'y-Coordinate:', Node(j,2))
end

% Read in nodal data

Node = zeros(nNode, 2); % Create node array

header = fgets(fid); % Skip over header lines
header = fgets(fid); % Skip over header lines

for i = 1:nNode
    j = fscanf(fid, '%d', 1); % Node number
    Node(j,1) = fscanf(fid, '%g', 1); % Nodal x-coordinate
    Node(j,2) = fscanf(fid, '%g
', 1); % Nodal y-coordinate

    if (iQuiet == 0)
        disp(sprintf('%s %d %s %6.2f %s %6.2f,...
', 'Node number: ', j,...
        'x-Coordinate: ', Node(j,1), ...
        'y-Coordinate: ', Node(j,2)))
    end
end

if iQuiet == 0
fprintf(1, 'n')
end

******************************************************************************
% Read in elemental connectivity data
******************************************************************************

Elem(1:nElem) = struct('nn',[-1 -1 -1],'edge',[-1 -1 -1],'stress',0.0,...
    'growth',0.0,'matl',-1,'area',0.0,'age',0);

header = fgets(fid);    % Skip over header lines
header = fgets(fid);    % Skip over header lines

for i = 1:nElem
    j = fscanf(fid,'%d',1);    % Element number
    Elem(j).nn(1) = fscanf(fid,'%d',1);    % Node 1
    Elem(j).nn(2) = fscanf(fid,'%d',1);    % Node 2
    Elem(j).nn(3) = fscanf(fid,'%d',1);    % Node 3
    Elem(j).matl = fscanf(fid,'%d',1);    % Material

    if iQuiet == 0
        fprintf(1,'%s%4d%s%4d%s%4d%s%2d
',
            'Element ',j,...
            'Node 1: ',Elem(j).nn(1),...
            'Node 2: ',Elem(j).nn(2),...
            'Node 3: ',Elem(j).nn(3),...
            'Material# ',Elem(j).matl);
    end
    Elem(j).area =
        ElemArea(Elem(j).nn(1),Elem(j).nn(2),Elem(j).nn(3),Node);
end

if iQuiet == 0
    fprintf(1,'n')
end

******************************************************************************
% Read in nodal constraints
******************************************************************************

Con(1:nCon) = struct('Node',0,'Dir',0);

header = fgets(fid);    % Skip over header lines
header = fgets(fid);    % Skip over header lines

for i = 1:nCon
    Con(i).Node = fscanf(fid,'%d',1);
    Con(i).Dir = fscanf(fid,'%d',1);

    if iQuiet == 0
        if Con(i).Dir == 1
            dirText = 'x';
        else
            dirText = 'y';
        end
    end
disp(sprintf('%s %4d %s %s','Constrained Node:
',Con(i).Node,...
Direction:
',dirText))
end

if iQuiet == 0
    fprintf(1,'
')
end

*******************************
% Read in symmetry data
*******************************

Sym = zeros(nSym,1);

header = fgets(fid); % Skip over header lines
header = fgets(fid); % Skip over header lines

for i = 1:nSym
    j = fscanf(fid, '%d',1); % Line number
    Sym(j) = fscanf(fid, '%d
',1); % Symmetry line direction
    if (iQuiet == 0)
        disp(sprintf('%s%d%s%f%s%f
' ,'Loaded Node ',Load(i).Node,
x-Force: ',Load(i).x,
y-Force: ',Load(i).y)
    end
end

*******************************
% Read in loads
*******************************

Load(1:nLoad) = struct('Node',0,'x',0.0,'y',0.0);

header = fgets(fid); % Skip over header lines
header = fgets(fid); % Skip over header lines

for i = 1:nLoad
    Load(i).Node = fscanf(fid, '%d',1);
    Load(i).x = fscanf(fid, '%g',1);
    Load(i).y = fscanf(fid, '%g
',1);
    if (iQuiet == 0)
        fprintf(1,'%s%d%s%f%s%f
','Loaded Node ',Load(i).Node,...
x-Force: ',Load(i).x,
y-Force: ',Load(i).y)
    end
end
if iQuiet == 0
    fprintf(1,'\n')
end

% Read in material properties
Prop = zeros(nMat,3); % Material properties (modulus of elasticity)

header = fgets(fid); % Skip over header lines
header = fgets(fid); % Skip over header lines

for i = 1:nMat
    iTemp(1) = fscanf(fid, '%d',1);
    ftemp = fscanf(fid, '%g%g%g\n',3);
    Prop(iTemp(1),1) = ftemp(1); % Modulus of elasticity
    Prop(iTemp(1),2) = ftemp(2); % Poisson's ratio
    Prop(iTemp(1),3) = ftemp(3); % Thickness

    if iQuiet == 0
        fprintf(1,'%s%g%g%g\n',...
        'Material#'
        ',iTemp(1),...
        ',E:
        ',Prop(iTemp(1),1),...
        ',Nu:
        ',Prop(iTemp(1),2),...
        ' Thickness: ',Prop(iTemp(1),3))
    end
end

if iQuiet == 0
    fprintf(1,'\n')
end

fclose(fid);

CSTSOLVER.m

% The program CSTSolver uses the Constant Strain Triangle method
% to solve a 2D plate problem.
% iFlag = 1 returns element stresses and leaves nodes unchanged
% iFlag = 2 returns nodal coordinates and leaves elements unchanged

function output = CSTSolver(Con,Node,Elem,Load,Prop,GrowLoad)

% Determine lengths of data structures
nLoad = length(Load);
nCon = length(Con);
nNode = length(Node);
nElem = length(Elem);

% Decide whether to output element stresses or nodal displacements
if nargin == 5
    iFlag = 1;  % Output element stresses
else
    iFlag = 2;  % Output nodal displacements
end

% Declare variables
i = 0;  % Integer index variable
j = 0;  % Integer index variable
k = 0;  % Integer index variable
c = zeros(3,1);  % Temporary floating point variable
n = zeros(3,1);  % Temporary integer variable
D = zeros(3,3);  % Stress/strain matrix
B = zeros(3,6);  % Strain/displacement matrix
SE = zeros(6,6);  % Elemental stiffness matrix
q = zeros(6,1);  % Elemental displacement vector

% Important constants
nDN = 2;  % Degrees of freedom per node
nDim = 2;  % Dimensions in problem
nDOF = nDN * nNode;  % Size of problem
S = zeros(nDOF,nDOF);  % Global stiffness matrix
Q = zeros(nDOF,1);  % Global displacement matrix
F = zeros(nDOF,1);  % Global load vector

% iQuiet = 1 suppresses output
iQuiet = 1;

if iQuiet == 0
    fprintf(1,'%s%4d
', 'Degrees of freedom : ',nDOF)
    fprintf(1,'
')
end

% Begin calculating global stiffness matrix
for i = 1:nElem
    % Elemental connectivity
    [n,B,D,t,DetJ] = DBMat(ElemNode,Prop,i);
    % Elemental stiffness matrix
    SE = 0.5 * t * DetJ * B' * D * B;
    % Put elements into global stiffness matrix
    j = 1:3;
    k = 1:3;
    S(2*n(j)-1,2*n(k)-1) = S(2*n(j)-1,2*n(k)-1)+SE(2*j-1,2*k-1);
    S(2*n(j)-1,2*n(k) ) = S(2*n(j)-1,2*n(k) )+SE(2*j-1,2*k ) ;
    S(2*n(j), 2*n(k)-1) = S(2*n(j), 2*n(k)-1)+SE(2*j ,2*k-1);
    S(2*n(j), 2*n(k) ) = S(2*n(j), 2*n(k) )+SE(2*j ,2*k ) ;
end
penalty = 10000.0 * max(max(S));

% Add constraint penalties to stiffness matrix
for i = 1:nCon
    if Con(i).Dir == 1
        n(1) = 2*Con(i).Node - 1;
    else
        n(1) = 2*Con(i).Node;
    end
    S(n(1),n(1)) = S(n(1),n(1)) + penalty;
end

if iFlag == 1
% Calculate load vector
    for i = 1:nLoad
        F(2*Load(i).Node - 1) = Load(i).x;
        F(2*Load(i).Node) = Load(i).y;
    end
else
% Use GrowLoad forces to "grow" elements
    for i = 1:nNode
        F(2*i - 1) = GrowLoad(i,1);
        F(2*i) = GrowLoad(i,2);
    end
% Fix nodes where actual loads are applied
    for i = 1:nLoad
        n(1) = 2*Load(i).Node - 1;
        S(n(1),n(1)) = S(n(1),n(1)) + penalty;
        n(1) = n(1) + 1;
        S(n(1),n(1)) = S(n(1),n(1)) + penalty;
    end
end
% Displacement vector
Q = inv(S)*F;

if iQuiet == 0
    for i = 1:nNode
        fprintf(1,'%s%3d%s%12.4g%s%12.4g
',
            'Node: ',i,...
            'x-Disp: ',Q(2*i-1),...
            'y-Disp: ',Q(2*i))
    end
end

if iFlag == 1
% Elemental stresses
    for i = 1:nElem
        ...
% Elemental displacement vector
q(2*j-1) = Q(2*n(j)-1);
q(2*j ) = Q(2*n(j) );

% Elemental stresses
s(:,i) = D*B*q;

% Stress invariants
I1 = s(1,i) + s(2,i);
I2 = s(1,i)*s(2,i) - s(3,i)*s(3,i);

% Von mises stress
Elem(i).stress = (I1*I1 - 3*I2)^0.5;
end
output = Elem;
else

% Nodal displacements
for i = 1:nNode
    Node(i,1) = Node(i,1) + Q(2*i - 1);
    Node(i,2) = Node(i,2) + Q(2*i);
end
output = Node;
end

******************************************************************************
% *** The function DBMat calculates D and B matrices for the constant
% *** strain triangle FEM problem
******************************************************************************

function [n,B,D,t,detJ] = DBMat(Elem,Node,Prop,i)

% Store node numbers
n(1) = Elem(i).nn(1);
n(2) = Elem(i).nn(2);
n(3) = Elem(i).nn(3);

% Material properties
E = Prop(Elem(i).matl,1);
Nu = Prop(Elem(i).matl,2);
t = Prop(Elem(i).matl,3);

% Calculate D() matrix
c(1) = E/(1 - Nu^2);
c(2) = c(1) * Nu;
c(3) = 0.5 * E/(1 + Nu);
D = [c(1) c(2) 0; c(2) c(1) 0; 0 0 c(3)];

% Store nodal coordinates
x21 = Node(n(2),1) - Node(n(1),1);
x32 = Node(n(3),1) - Node(n(2),1);
x13 = Node(n(1),1) - Node(n(3),1);
y12 = Node(n(1),2) - Node(n(2),2);
y23 = Node(n(2),2) - Node(n(3),2);
y31 = Node(n(3),2) - Node(n(1),2);
detJ = x13 * y23 - x32 * y31;

% Calculate B() matrix
B = (1/detJ)*[y23 0 y31 0 y12 0;
0 x32 0 x13 0 x21;
x32 y23 x13 y31 x21 y12];

PLATEPLOT.m

% Function PlatePlot plots triangular CST elements
function PlatePlot (Node, Elem, Edge, Con, Load)

% Compute maximum and minimum axis limits
nNode = length(Node);
nElem = length(Elem);
nEdge = length(Edge);
nCon = length(Con);
nLoad = length(Load);

% Calculate the limits and lengths of the plot area
xMax = max([Node(:,1)]);
xMin = min([Node(:,1)]);
yMax = max([Node(:,2)]);
yMin = min([Node(:,2)]);

xSize = xMax - xMin;
ySize = yMax - yMin;

% Gives the plot area a boarder
margin = 0.1 * (max([xSize ySize]));

% Resets the plot area limits incorporating the margins
xMax = xMax + margin;
xMin = xMin - margin;
yMax = yMax + margin;
yMin = yMin - margin;

%figure
axis([xMin xMax yMin yMax])
%axis off
hold on

Stress = BestFit(Elem,nElem,Node,nNode);
for i = 1:nElem
    Connect(i,:) = [Elem(i).nn(1) Elem(i).nn(2) Elem(i).nn(3)];
end

for i = 1:nNode
    Vertex(i,:) = [Node(i,1) Node(i,2)];
end

% Here we add an extra element to the connectivity matrix.
% This is a workaround to ensure that the number of elements & nodes
% are not equal. If they are equal, Matlab does not plot interpolated colors correctly. This is cheesy, but effective.
if nElem == nNode
    Connect(nElem + 1,:) = Connect(nElem,:);
end

% Plot triangular elements
patch('Vertices',Vertex,'Faces',Connect,'FaceVertexCData',Stress,...
    'FaceColor','interp','EdgeColor','none');

% Add scale to plot
sMax = max(Stress);
sMin = min(Stress);
for i = 0:5
    lText = int2str((i/5)*(sMax - sMin) + sMin);
    text(xMax,(i/10)*yMax+0.5*yMax,lText);
end

% Plot element numbers
for i = 1:nElem
    x = (sum([Node(Elem(i).nn,l)])/3);
    y = (sum([Node(Elem(i).nn,2)])/3);
    text(x,y,int2str(i), 'Color', 'k','HorizontalAlignment','center')
end

% Plot node numbers
for i = 1:nNode
    text(Node(i,1),Node(i,2),int2str(i), 'Color', 'k',...
        'HorizontalAlignment','center')
end

% Plot border elements
for i = 1:nEdge
    x = [Node(Edge(i).nn(1),l) Node(Edge(i).nn(2),l)];
    y = [Node(Edge(i).nn(1),2) Node(Edge(i).nn(2),2)];
    xmid = (Node(Edge(i).nn(1),1) + Node(Edge(i).nn(2),1))/2;
    ymid = (Node(Edge(i).nn(1),2) + Node(Edge(i).nn(2),2))/2;
    if Edge(i).elem(2) == 0
        plot(x,y,'b','LineWidth',2)
    else
        plot(x,y,':k')
    end
    text(xmid,ymid,int2str(i), 'Color','k','HorizontalAlignment','center')
end

% Dimensions of triangle for displaying loads & constraints
height = 0.03 * (xMax - xMin);
width = 0.3 * height;

% Plot nodal constraints
for i = 1:nCon
function F = BestFit(Elem,nElem,Node,nNode)

Global "stiffness matrix"

Global "load vector"

Vector of nodal values

Begin calculating global stiffness matrix

% Declare variables
i = 0; % Integer index variable
j = 0; % Integer index variable
k = 0; % Integer index variable
n = zeros(3,1); % Temporary integer variable
RE = zeros(3,1); % Strain/displacement matrix
WE = zeros(3,3); % Nodal distribution matrix for element
W = zeros(nNode,nNode); % Global "stiffness matrix"
R = zeros(nNode,1); % Global "load vector"
F = zeros(nNode,1); % Vector of nodal values

% Begin calculating global stiffness matrix

\[
x = \text{Node}(\text{Con}(i).\text{Node}, 1);
y = \text{Node}(\text{Con}(i).\text{Node}, 2);
\]
\[
\text{if } \text{Con}(i).\text{Dir} == 1
\]
\[
\text{Vertex} = [x \ y; \ x-\text{height} \ y-\text{width}; \ x-\text{height} \ y+\text{width}];
\]
\[
\text{else}
\text{Vertex} = [x \ y; \ x+\text{width} \ y-\text{height}; \ x-\text{width} \ y-\text{height}];
\]
\[
\text{end}
\]

\[
\text{Connect} = [1 \ 2 \ 3];
\]

\[
\text{patch('Vertices',Vertex,'Faces',Connect,'FaceColor','k')}
\]

\[
\text{end}
\]

% Plot nodal loads

\[
\text{height} = 0.04 \times (\text{xMax} - \text{xMin});
\]

\[
\text{width} = 0.3 \times \text{height};
\]

\[
\text{maxLoad} = \max([\text{abs}(\text{Load}.x) \ \text{abs}(\text{Load}.y)]);
\]

\[
\maxLength = 0.07 \times (\text{xMax} - \text{xMin});
\]

\[
\text{for } i = 1:\text{nLoad}
\text{x} = \text{Node}(\text{Load}(i).\text{Node}, 1);
\text{y} = \text{Node}(\text{Load}(i).\text{Node}, 2);
\text{if } \text{Con}(i).\text{Dir} == 1
\text{Vertex} = [x \ y; \ x-\text{height} \ y-\text{width}; \ x-\text{height} \ y+\text{width}];
\text{else}
\text{Vertex} = [x \ y; \ x+\text{width} \ y-\text{height}; \ x-\text{width} \ y-\text{height}];
\text{end}
\text{Connect} = [1 \ 2 \ 3];
\text{patch('Vertices',Vertex,'Faces',Connect,'FaceColor','b')}
\text{xPoint} = [x x];
\text{yPoint} = [y y + 2*\text{height}];
\text{plot(xPoint,yPoint,'b','LineWidth',2)
\text{end}
\]

\[
\text{hold off}
\]

*** The program BestFit finds nodal values of stresses for CST elements for creating contour plots.*****************
for i = 1:nElem

% Store node numbers
n(1) = Elem(i).nn(1);
n(2) = Elem(i).nn(2);
n(3) = Elem(i).nn(3);

% Store nodal coordinates
x23 = Node(n(2),1) - Node(n(3),1);
x31 = Node(n(3),1) - Node(n(1),1);
y23 = Node(n(2),2) - Node(n(3),2);
y31 = Node(n(3),2) - Node(n(1),2);

detJ = x23 * y31 - x31 * y23;

% Calculate nodal distribution matrix for element
WE = (detJ/24)*[2 1 1; 1 2 1; 1 1 2];

% Calculate elemental "load" vector
RE = abs(Elem(i).stress) * (detJ/6)*[1; 1; 1];

% Put elements into global stiffness matrix
j = 1:3;
k = 1:3;
W(n(j),n(k)) = W(n(j),n(k)) + WE(j,k);

% Assemble "load" vector
R(n(j)) = R(n(j)) + RE(j);
end

F = inv(W)*R;

**NEWELEMENT.m**

*****************************************************************************
% The program, NewElem, adds a new element onto an existing element
% that has the capacity to grow an element from a free edge. The new
% element has the same area as the existing element unless there is a
% close enough node to connect to.
*****************************************************************************

function [Node,Elem,Edge,Con] = NewElem(Node,Elem,Edge,Con,Load,Prop,DeathYear,year,NewStressMax)

% Determine the number of nodes and elements within each data structure
nElem = length(Elem);
nOldNode = length(Node);

% Solve for stresses in elements
Elem = CSTSolver(Con,Node,Elem,Load,Prop);

% Calculate maximum allowable stress in each element before growth
NewStress = (tanh(((8/DeathYear)*year)-4)+1)*(NewStressMax/2);

% Loop through elements to see which are subject to growth
% GrowElem, Node, nGrowElem = MakeGrowElem(Node, Elem, Edge, NewStress);

% Loop through newly added nodes to see if any can be merged
GrowElem, Node = MergeNewNodes(GrowElem, Node);

% Loop through newly added and existing nodes to see if any can be merged
GrowElem, Node = MergeOldNodes(GrowElem, Node, nOldNode);

% Loop through proposed elements to eliminate any duplicates
GrowElem = MergeNewElems(GrowElem);

nGrowElem = length(GrowElem);

% Add elements to the structure
for i = 1:nGrowElem
    nElem = nElem + 1;

    % Node numbers of growth element
    n(1) = GrowElem(i).nn(1);
    n(2) = GrowElem(i).nn(2);
    n(3) = GrowElem(i).nn(3);
    nE = GrowElem(i).elem;

    % Loop through proposed edges to eliminate any duplicates
    [Edge, Elem] = MergeNewEdges(Edge, n, m, nElem, GrowElem(i).edge(1));

    % Add new element to structure
    Elem(nElem).nn = GrowElem(i).nn;
    Elem(nElem).stress = 0.0;
    Elem(nElem).matl = Elem(nE).matl;
    Elem(nElem).age = 0;

    % Calculate elemental area
    Elem(nElem).area = ElemArea(n(1), n(2), n(3), Node);
end

% Add constraints where nodes are along an axis of symmetry
% Con = AddConstraint(Con, Sym, Node);

% *** The function MakeGrowElem creates the GrowElem structure, a list
% *** of elements to be added along the boundaries of the structure.


nElem = length(Elem);
nodeName = length(Node);
nGrowElem = 0;
GrowElem = [];

for i = 1:nElem
    if abs(Elem(i).stress) > NewStress

% Loop through edges to find border edges
for j = 1:3
    nGE = Elem(i).edge(j); % Growth edge number
    if Edge(nGE).elem(2) == 0
        n1 = Edge(nGE).rn(1); % Growth edge node 1
        n2 = Edge(nGE).nn(2); % Growth edge node 2
        xl = Node(n1,:); % Coords of node 1
        x2 = Node(n2,:); % Coords of node 2
        x0 = mean([xl ; x2]); % Midpoint of growth edge
        p = x2 - xl; % Vector along growth edge
        n = [p(2) - p(1)]; % Vector normal to edge
        S = 2 * Elem(i).area / dot(p,p); % Scale factor for new node
        x4 = x0 + S*n; % Coordinates of new node
        [x4,iFlag] = CheckSym(x0,x4,Sym);
        iFlag = 0;
        if iFlag == 0
            nGrowElem = nGrowElem + 1;
            nNode = nNode + 1;
            Node(nNode,:) = x4;
        end
    end
end
end
end

%***********************************************************************
% *** The function MergeNewNodes finds newly-added nodes that are close
% *** together and merges them to eliminate duplicates.
%***********************************************************************

function [GrowElem,Node] = MergeNewNodes(GrowElem,Node)

    nGrowElem = length(GrowElem);

    for i = 1:nGrowElem
        n1 = GrowElem(i).nn(2); % Newly-added node 1
        for j = i+1:nGrowElem
            n2 = GrowElem(j).nn(2); % Newly-added node 2
            % Distance between newly-added nodes
            dist = pdist([Node(n1,:);Node(n2,:)]);
            EdgeMin = 0.55;
            % If distance is smaller than minimum edge size then merge nodes
            if dist < EdgeMin
                Node(n1,:) = mean([Node(n1,:);Node(n2,:)]);
                [GrowElem,Node] = MergeNodes(GrowElem,Node,j,n1,n2);
            end
        end
    end
end
The function MergeOldNodes finds newly-added nodes that are close to existing nodes and eliminates them to avoid duplicate nodes. Before merging nodes, the function calculates the area of the element with the proposed node to find out if the node is on the correct side of the growth edge. If it's not, the resulting area will be negative.

```matlab
function [GrowElem, Node] = MergeOldNodes (GrowElem, Node, nOldNode)

EdgeMin = 0.55;

for i = 1:length(GrowElem)
    nNew = GrowElem(i).nn(2); % Newly-added node
    minDist = 10e6; % Store distance to closest node
    mNode = 0; % Node number of closest node

    % Loop through existing nodes
    for nOld = 1:nOldNode
        dist = pdist([Node(nNew, :) ;Node(nOld,:)]); % Dist between nodes
        if dist < EdgeMin
            p = ElemArea(GrowElem(i).nn(1), nOld, GrowElem(i).nn(3), Node);
            if p > 0 & dist < minDist % Keep track of closest node
                mNode = nOld;
            end
        end
    end

    % If there is a close existing node, merge the new one into it
    if mNode ~= 0
        [GrowElem, Node] = MergeNodes(GrowElem, Node, i, mNode, nNew);
    end
end

% *** The function MergeNodes merges two nodes that are close together.
% *** nE = newly grown element number
% *** n1 = node to keep
% *** n2 = node to eliminate

function [GrowElem, Node] = MergeNodes (GrowElem, Node, nE, n1, n2)

GrowElem(nE).nn(2) = n1; % Change growth node to node n1
Node(n2,:) = []; % Eliminate node n2

% Decrement growth node numbers by one
for i = 1:length(GrowElem)

end
end
```

% *** The function MergeOldNodes finds newly-added nodes that are close to existing nodes and eliminates them to avoid duplicate nodes. Before merging nodes, the function calculates the area of the element with the proposed node to find out if the node is on the correct side of the growth edge. If it's not, the resulting area will be negative.
if GrowElem(i).nn(2) > n2
    GrowElem(i).nn(2) = GrowElem(i).nn(2) - 1;
end
end

*****************************************************************************
% *** The function MergeNewElems loops through the proposed elements to
% *** make sure that there are no duplicates.
*****************************************************************************

function GrowElem = MergeNewElems(GrowElem)
    nGrowElem = length(GrowElem);
    i = 1;
    while i <= nGrowElem
        j = i + 1;
        while j <= nGrowElem
            if sort(GrowElem(i).nn) == sort(GrowElem(j).nn)
                GrowElem(j) = [];
                nGrowElem = nGrowElem - 1;
            end
            j = j + 1;
        end
        i = i + 1;
    end

*****************************************************************************
% *** The function MergeNewEdges loops through the proposed new edges
% *** to make sure that there are no duplicates.
*****************************************************************************

function [Edge,Elem] = MergeNewEdges(Edge,Elem,n,nElem,nGE)
    nEdge = length(Edge);
    Elem(nElem).edge(1) = nGE; % 1st edge in new elem is growth edge
    Edge(nGE).elem(2) = nElem; % 2nd element is newly added element
    for i = 1:2
        iFlag = 0;
        for j = 1:nEdge
            if (Edge(j).nn(2) == n(i) & Edge(j).nn(1) == n(i+1))
                Edge(j).elem(2) = nElem;
                Elem(nElem).edge(i+1) = j;
                iFlag = 1;
                break
            end
        end
    end
    if iFlag == 0
        nEdge = nEdge + 1;
        Edge(nEdge).nn(1) = n(i);
        Edge(nEdge).nn(2) = n(i+1);
        Edge(nEdge).elem(1) = nElem;
        Edge(nEdge).elem(2) = 0;
        Elem(nElem).edge(i+1) = nEdge;
function [x4,iFlag] = CheckSym(x0,x4, Sym)

iFlag = 0;
for i = 1:length(Sym)
  if Sym(i) == 1
    if x0(1) == 0
      iFlag = 1;
      return
    elseif x4(1) < 0
      x4(1) = 0;
      x4(2) = (x4(1)*x0(2) - x0(1)*x4(2))/(x4(1)-x0(1));
    end
  else
    if x0(2) == 0
      iFlag = 1;
      return
    elseif x4(2) < 0
      x4(1) = (x0(1)*x4(2)-x4(1)*x0(2))/(x4(2)-x0(2));
      x4(2) = 0;
    end
end
end

function Con = AddConstraint(Con, Sym, Node);

nSym = length(Sym);
nNode = length(Node);
nCon = length(Con);

for i = 1:nNode
  x = Node(i,:);
  for j = 1:nSym
    if Sym(j) == 1
      q = [-1 0];
    else
      q = [0 -1];
    end
    if dot(q,x) == 0
      iFlag = 0;
      for k = 1:nCon
        if i == Con(k).Node & Sym(j) == Con(k).Dir
          iFlag = 1;
        end
      end
    end
  end
end
end
if iFlag == 0
nCon = nCon + 1;
Con(nCon).Node = i;
Con(nCon).Dir = Sym(j);
end
end
end

GROWELEM.m

******************************************************************************
% *** The program GrowElem expands the area of an existing interior
% *** element by moving its nodes outward.
******************************************************************************

function Node = GrowElem(Con, Node, Elem, Load, Prop, DeathYear, year, GrowStressMax, factor);

% Determine lengths of data structures
nLoad = length(Load);
nCon = length(Con);
nNode = length(Node);
nElem = length(Elem);

GrowStress = (tanh(((8/DeathYear)*year)-4)+1)*(GrowStressMax/2);
GrowLoad = zeros(nNode,2);

% Growth factor
iDOF = 0;

for i = 1:nElem

    % If stress is too big, grow element
    if Elem(i).stress > GrowStress

        xCent = 0; % Centroid of element
        yCent = 0; % Centroid of element

        % Elemental nodes
        for j = 1:3
            n(j) = Elem(i).nn(j);
            x(j) = Node(n(j),1);
            y(j) = Node(n(j),2);
            xCent = xCent + x(j);
            yCent = yCent + y(j);
        end

        % Centroid of element
        xCent = xCent/3;
        yCent = yCent/3;

        % Nodal displacements
end
end
end
end
end
GROWELEM.m
for j = 1:3
    iflag = 0;
    for k = 1:nLoad
        if n(j) == Load(k).Node
            iflag = 1;
        end
    end
    for k = 1:nCon
        if n(j) == Con(k).Node
            iflag = 1;
        end
    end
    if iflag == 0
        fX = factor*(x(j) - xCent);
        fY = factor*(y(j) - yCent);
        nN = Elem(i).nn(j);
        GrowLoad(nN, 1) = GrowLoad(nN, 1) + fX;
        GrowLoad(nN, 2) = GrowLoad(nN, 2) + fY;
    end
end
end
end
end
if abs(max(max(GrowLoad))) ~= 0
    Node = CSTSolver(Con,Node, Elem, Load, Prop, GrowLoad);
end

KILLELEM.m

**************************************************************************************
% *** The program KillElem eliminates elements with stresses less than the absolute value of the minimum stress. Working in conjunction with the NewElem.m routine, this will not kill elements that were just created.
**************************************************************************************

function [Node, Elem, Edge, Load, Con] = KillElem(Node, Elem, Edge, Con, Load, Prop, DeathYear, RemStressMax, year)
% Determine the number of elements within structure
nElem = length(Elem);

% Set the minimum stress for the element to live
RemStress = (tanh(((8/DeathYear)*year)-4)+1)*(RemStressMax/2);

% Make a list of elements for removal, sorted by stress level
[RemElem, nRemElem] = MakeRemElem(Elem, RemStress, nElem, Load, Con);

% Loop through the elements to be removed
for i = 1:nRemElem
    nE = RemElem(i); % Number of element to be removed

    % Count the number of border edges on the element
    BorderEdge = CountBorder(nE, Elem, Edge);
% The procedure for element removal depends upon whether the element is
% an interior element, or has one or two border edges.

switch length(BorderEdge)

% *** Interior element
% *** Remove element only
    case 0
        DOF = CheckMobility(nE,Elem,Node,Con);
        if DOF > 0 % If removal of element creates a
            continue % mechanism, then skip it
        end
        [Edge] = CheckInteriorEdges(Elem,Edge,nE);
        Elem(nE) = []; % Remove element
        [Edge] = DecEdgeElem(nE,Edge); % Decrement element numbers
                        % in Edge

% *** Element with one border edge
% *** Remove element and border edge
    case 1
        DOF = CheckMobility(nE,Elem,Node,Con);
        if DOF > 0 % If removal of element creates a
            continue % mechanism, then skip it
        end
        [Edge] = DecEdgeElem(nE,Edge); % Decrement element numbers
                        % in Edge
        Elem(nE) = []; % Remove element
        nG = BorderEdge(1);
        [Elem] = DecElemEdge(nG,Elem); % Decrement edge numbers in
                                        % Elem
        Edge(nG) = []; % Remove border edge

% *** Element with two border edges
% *** Remove element, two border edges and possibly a node
    case 2 % Element with two border
        nG1 = max(BorderEdge); % Start with higher edge
        nG2 = min(BorderEdge); % Finish with lower edge

        nN = intersect(Edge(nG1).nn,Edge(nG2).nn);
        m = length(find([Elem.nn] == nN));
        DOF = CheckMobility(nE,Elem,Node,Con) - 2*(2-m);
        if DOF > 0 % If removal of element creates a
                    continue % mechanism, then skip it
        end

        if m == 1
            Node(nN,:) = [];
[Edge] = DecEdgeNode(nN, Edge); % Decrement node #s in
% Edge
[Elem, Load, Con] = DecElemNode(nN, Elem, Load, Con); % Decrement node #s in
% Elem
end

[Edge] = DecEdgeElem(nE, Edge); % Decrement element #s in
% Edge

Edge(nGl) = []; % Remove first border edge
[Elem] = DecElemEdge(nGl, Elem); % Decrement edge #s in
% Elem

Edge(nG2) = []; % Remove second border edge
[Elem] = DecElemEdge(nG2, Elem); % Decrement edge #s in
% Elem

Elem(nE) = []; % Remove element

for j = i:nRemElem % Decrement element #s in RemElem
    if RemElem(j) > RemElem(i)
        RemElem(j) = RemElem(j) - 1;
    end
end
end

******************************************************************************
** The function MakeRemElem creates the RemElem array, a list of
** elements to be removed. Only elements with age > 1 are eligible
** for removal.
******************************************************************************

function [RemElem, nRemElem] = MakeRemElem(Elem, RemStress, nElem, Load, Con)

nLoad = length(Load);
 nCon = length(Con) ;
 LC = nLoad + nCon;
 Load_Con = zeros(LC, 1);

NotKill = 0;
 nRemElem = 0;
 RemElem = [];

for k = 1:nLoad
    Load_Con(k) = Load(k).Node;
end
for n = 1:nCon
    Load_Con(nLoad+n) = Con(n).Node;
end

% Sort elements in ascending stress order
[stress, index] = sortrows([Elem.stress]);

for i = 1:nElem
    j = index(i);

    for k = 1:LC
        for m = 1:3

******************************************************************************
if Elem(j).nn(m) == Load_Con(k)
    NotKill = 1;
    break
end
end
if NotKill == 1
    break
end
end
if abs(Elem(j).stress) < RemStress & Elem(j).age > 0 & NotKill == 0
    nRemElem = nRemElem + 1;
    RemElem(nRemElem) = j;
    NotKill = 0;
end
end

% *** The function CountBorder counts the number of exterior edges on an element
function BorderEdge = CountBorder(nE, Elem, Edge)
    BorderEdge = [];
    j = 0;
    for i = 1:3
        nG = Elem(nE).edge(i);
        if Edge(nG).elem(2) == 0
            j = j + 1;
            BorderEdge(j) = nG;
        end
    end
end

% *** The function DecEdgeElem reduces the element numbers in the Edge array by 1 where they are greater than the number of a deleted element.
function [Edge] = DecEdgeElem(nE, Edge)
    for i = 1:length(Edge)
        % If the first entry in Edge is nE, then replace the first entry with the second entry and make the second entry 0, thus creating a border edge.
        if Edge(i).elem(1) == nE & Edge(i).elem(2) < nE
            Edge(i).elem(1) = Edge(i).elem(2);
            Edge(i).elem(2) = 0;
        elseif Edge(i).elem(1) == nE & Edge(i).elem(2) > nE
            Edge(i).elem(1) = Edge(i).elem(2) - 1;
            Edge(i).elem(2) = 0;
        % If the first entry in Edge is greater than nE, then decrement it by 1
        elseif Edge(i).elem(1) > nE
            Edge(i).elem(1) = Edge(i).elem(1) - 1;
    end
end
% If the second entry in Edge is nE, then replace it with zero, thus % creating a border edge.
if Edge(i).elem(2) == nE
    Edge(i).elem(2) = 0;
    display(['Edge 2 is removed elem'])
end

% If the second entry in Edge is greater than % nE, then decrement it by 1
elseif Edge(i).elem(2) > nE
    Edge(i).elem(2) = Edge(i).elem(2) - 1;
end

% *** The function DecEdgeNode reduces the node numbers in the Edge % *** array by 1 where they are greater than the number of a deleted % *** node.
***********************************************************************
function [Edge] = DecEdgeNode(nN, Edge)
    for i = 1:length(Edge)
        for j = 1:2
            if Edge(i).nn(j) > nN
                Edge(i).nn(j) = Edge(i).nn(j) - 1;
            end
        end
    end

% *** The function DecElemEdge reduces the edge numbers in the Elem % *** array by 1 where they are greater than the number of a deleted % *** edge.
***********************************************************************
function [Elem] = DecElemEdge(nG, Elem)
    for i = 1:length(Elem)
        for j = 1:3
            if Elem(i).edge(j) >= nG
                Elem(i).edge(j) = Elem(i).edge(j) - 1;
            end
        end
    end

% *** The function DecElemNode reduces the node numbers in the Elem % *** array by 1 where they are greater than the number of a deleted % *** node.
***********************************************************************
function [Elem, Load, Con] = DecElemNode(nN, Elem, Load, Con)
nLoad = length(Load);
nCon = length(Con) ;
for i = 1:length(Elem)
    for j = 1:3
        if Elem(i).nn(j) > nN
            Elem(i).nn(j) = Elem(i).nn(j) - 1;
        end
    end
end

for k = 1:nLoad
    if Load(k).Node > nN
        Load(k).Node = Load(k).Node - 1;
    end
end

for k = 1:nCon
    if Con(k).Node > nN
        Con(k).Node = Con(k).Node - 1;
    end
end

***********************************************************************
% *** The function CheckMobility determines whether or not the
% *** structure would be mobile upon the removal of an element.
%***************************************************************************

function DOF = CheckMobility(nE,Elem,Node,Con)

% Initialize the "strain-displacement" matrix
S = zeros(3*length(Elem)-1+length(Con),2*length(Node));

% Loop through elements
for i = 1:length(Elem)
    if i == nE
        continue
    end
    j = 1:3;
    k = 1:3;
    m = [2 3 1];
    n(j) = Elem(i).nn(j);
    dx(j) = Node(n(j),1)-Node(n(m),1);
    dy(j) = Node(n(j),2)-Node(n(m),2);

    % Create elemental strain-displacement matrix
    B = [ dy(2) 0 dy(3) 0 dy(1) 0 ;
          0 -dx(2) 0 -dx(3) 0 -dx(1) ;
          -dx(2) dy(2) -dx(3) dy(3) -dx(1) dy(1) ];

    % Add into global matrix
    S(3*(i-1)+k,2*n(j)-1) = S(3*(i-1)+k,2*n(j)-1)+B(k,2*j-1);
    S(3*(i-1)+k,2*n(j)) = S(3*(i-1)+k,2*n(j)) + B(k,2*j);
end

% Add contributions from constraints
for i = 1:length(Con)
    S(3*length(Elem)+i,2*(Con(i).Node-1) + Con(i).Dir) = 1;
end
% Calculate mobility based upon rank of strain-displacement matrix
DOF = 2*length(Node)- rank(S);

***********************************************************************
*** The function CheckInteriorEdges determines whether or not the
*** edges' nodes are assigned properly for creating new elements
*** to fill the void
***********************************************************************

function [Edge] = CheckInteriorEdges(Elem, Edge,nElem)
nEdge = length(Edge);
for i = 1:3
    nEdge = Elem(nElem).edge(i);
    % Checking each edge
    % the edges of the removed
    % element
    for j = 1:3
        k = j + 1;
        if k == 4
            k = 1;
        end
        % if the nodes in Edge are in the same order as the nodes in
        % Elem a new element will grow in the wrong direction
        if (Edge(nEdge).nn(1) == Elem(nElem).nn(j) & Edge(nEdge).nn(2) == Elem(nElem).nn(k))
            % x = Edge(nEdge).nn(1);
            % store the
            % number of node 1
            % Edge(nEdge).nn(1) = Edge(nEdge).nn(2); % replace node 1
            % with node 2
            % Edge(nEdge).nn(2) = x; % replace node 2
            % with node 1
            % Edge(nEdge).NewElem = 0;
            else
                Edge(nEdge).NewElem = 1;
            end
        end
    end
end

GETMASS.m

function mass = GetMass(Elem,Prop,StressMax);

mass = 0;
nElem = length(Elem);
penalty = 10000; % Penalty for the stress in one element being too high

for i = 1:nElem
    if Elem(i).stress <= StressMax
        mass = mass + Elem(i).area * Prop(Elem(i).matl,3) * Prop(Elem(i).matl,4);
    else
mass = mass + Elem(i).area * Prop(Elem(i).matl,3) *
Prop(Elem(i).matl,4) * penalty;
end
end

ELEMAREA.m

**************************************************************************
% *** Function ElemArea - calculates area of a triangular element
% *** Input: three node numbers and the nodal coordinates array
% *** Output: elemental area
**************************************************************************
function ElemArea = ElemArea(n1,n2,n3,Node)
x1 = Node(n1,1);
x2 = Node(n2,1);
x3 = Node(n3,1);
y1 = Node(n1,2);
y2 = Node(n2,2);
y3 = Node(n3,2);
ElemArea = 0.5*((x2 - x3)*(y3 - y1) - (x3 - x1)*(y2 - y3));

EDGEDETECT.m

**************************************************************************
% *** The program EdgeDetect determines which edges of the triangular
% *** elements are on the outside border of the structure. The first
% *** two fields are the node numbers of an edge, the final two fields
% *** determine which element(s) the edge belongs to. If an edge
% *** belongs to only one element (i.e. if it is a border element) then
% *** the second Elem field contains a zero.
**************************************************************************
function [Elem,Edge] = EdgeDetect(Elem)

% Determine the number of elements in the data structure
nElem = length(Elem);

Edge = struct('nn',[0,0],'elem',[0,0]); %Create Edge structure
nEdge = 0;
% This loop encompasses the entire data structure
for i = 1:nElem
    % This loop goes to each 'first' node of the 'first' edge
    for j1 = 1:3
        j2 = j1 + 1;
        % This loop goes to each 'second' node of the 'first' edge
        if j2 == 4
            j2 = 1;
        end
        % Sets the current 'first' node of the current element to n1
        n1 = Elem(i).nn(j1);
        % Sets the current 'second' node of the current element to n2

n2 = Elem(i).nn(j2);

iFlag = 0;
for k = 1:nEdge
    if (Edge(k).nn(2) == ni & Edge(k).nn(1) == n2)
        iFlag = 1;
        Edge(k).elem(2) = i;
        Elem(i).edge(j1) = k;
        break
    end
end

if iFlag == 0
    nEdge = nEdge + 1;
    Edge(nEdge).nn(1) = ni;
    Edge(nEdge).nn(2) = n2;
    Edge(nEdge).elem(1) = i;
    Edge(nEdge).elem(2) = 0;
    Edge(nEdge).NewElem = 1;
    Elem(i).edge(j1) = nEdge;
end
end

EDGELENGTH.m

*******************************************************************************
% *** Function EdgeLength - calculates the distance between two nodes
% *** Input: two node numbers and the nodal coordinates array
% *** Output: distance between nodes
*******************************************************************************

function EdgeLength
    = EdgeLength(n1,n2,Node)

d = Node(n1,:) - Node(n2,:);
EdgeLength = sqrt(dot(d,d));

WRITEBRIDGE.m

function tracker =
WriteBridge(Node,Elem,Edge,Con,Load,trackgen,trackbridge,z)

nNode = length(Node);
nElem = length(Elem);
nCon = length(Con);
nLoad = length(Load);
nEdge = length(Edge);
extension = ['G' int2str(trackgen) '_B' int2str(trackbridge) '_Y' int2str(z) '.txt'];
filename = fullfile('C:','Documents and Settings','keith','Desktop','GA_MD','Genetic Algorithm','Data Bank',extension);
fid = fopen(filename,'w+');
% Information at the top of the text file
fprintf(fid,'%s
','***************************************************
**********');
fprintf(fid,'%s
','<< Bridge Information for Plotting >>');
fprintf(fid,'%s%d
','Generation :',trackgen);
fprintf(fid,'%s%d
','Bridge :',trackbridge);
fprintf(fid,'%s%d
','Year :',z);
fprintf(fid,'%s		%s		%s		%s		%s
','nNode','nElem','nCon','nLoad','nEdge');
fprintf(fid, '%d			%d			%d			%d
',nNode,nElem,nCon,nLoad,nEdge);
fprintf(fid,'%s
','***************************************************
**********');

% Information about the Nodes
LNode = length(Node);
fprintf(fid,'%s%s
','Node','#','X','Y');
for i = 1:LNode
    fprintf(fid, '%d		%g		%g
',i,Node(i,1),Node(i,2));
end
fprintf(fid,'%s
','***************************************************
**********');

% Information about the elements
LElem = length(Elem);
fprintf(fid,'%s%s
','Elem','#','N1','N2','N3','Stress');
for i = 1:LElem
    fprintf(fid, '%d		%d		%d		%d		%g
',i,Elem(i).nn(1),Elem(i).nn(2),Elem(i).nn(3),Elem(i).stress);
end
fprintf(fid,'%s
','***************************************************
**********');

% Information about the constraints
LCon = length(Con);
fprintf(fid,'%s%s
','Constrained Node','Direction (x=1,y=2)');
for i = 1:LCon
    fprintf(fid, '%d		%d
',Con(i).Node,Con(i).Dir);
end
fprintf(fid,'%s
','***************************************************
**********');

% Information about the loads
LLoad = length(Load);
fprintf(fid,'%s%s
','Load Node','X','Y');
for i = 1:LLoad
    fprintf(fid, '%d		%d		%g
',Load(i).Node,Load(i).x,Load(i).y);
end
fprintf(fid,'%s
','***************************************************
**********');
% Information about the edges

LEdge = length(Edge);
fprintf(fid,'%s		%s		%s		%s
','Edge #','Node 1','Node 2','Border?');
for i = 1:LEdge
    fprintf(fid,'%d		%d		%d		%d
',i,Edge(i).nn(1),Edge(i).nn(2),Edge(i).elem(2));
end
fclose(fid);

tracker = [trackgen,trackbridge,z];

GENESIS.m

function p = Genesis(pop,nGene)
for z = 1:pop
    p(z).genes = round(rand(1,nGene));
end

SET_OBJ.m

function obj_fxn = Set_Obj(nGene,chunks,lenGene)
    obj_bin = round(rand(1,nGene));
    obj_fxn = zeros(1,chunks);
    for j = 1:chunks
        for m = 1:lenGene
            n = m + (j-1)*lenGene;
            r = lenGene - m;
            obj_fxn(j) = obj_fxn(j) + obj_bin(n)*2^(r);
        end
    end

OBJECTIVE.m

% This subroutine will determine which structure has the highest mass. % The other structures in the current generation will be judged based % on the worst structure in the class. The fitness is stored in % p.fitness.
function p = Objective(p,pop)
for i = 1:pop
    mass(i) = p(i).mass;
end
for k = 1:pop
    p(k).fitness = max(mass) - p(k).mass;
end

CONVERT GENES.m

% This subroutine converts the binary data to actual numbers for use in
% the program. It uses basic interpolation techniques.

function p = Convert_Genes(pop,nGene,lenGene,p,chunks,range)
    for i = 1:pop
        for j = 1:chunks
            p(i).tempvar(j) = 0;
            for m = 1:lenGene
                n = m + (j-1)*lenGene;
                r = lenGene - m;
                p(i).tempvar(j) = p(i).tempvar(j) + p(i).genes(n)*2^r;
            end
        end
    end
    % Interpolate to find the actual value of the binary number in the range
    % that it is allowed to be in.
    binaryhigh = 2^(lenGene+1)-1;
    for i = 1:pop
        for j = 1:chunks
            p(i).variable(j) = round(range(j,2)-((range(j,2)-range(j,1))*((binaryhigh-p(i).tempvar(j))/binaryhigh)));
        end
    end

ROULETTE.m

% Randomly selects a candidate based on weighted optimality

function winner = Roulette(p,pop)
    rpop = pop - 1;
    for w = 1:rpop
        fitness(w) = p(w).fitness;
    end
    base = sum(fitness);
    [order,swap] = sort(fitness);
    wheel = order/base;
    for r = 2:rpop
        wheel(r) = wheel(r-1) + wheel(r);
    end
    hit = rand;
for t = 1:rpop
    if hit < wheel(t)
        x = swap(t);
        break
    else
        x = swap(rpop);
    end
end
winner.genes = p(x).genes;

LAYOVER.m

% Selects the keepers from the current generation to live into the next % generation.
function child = Layover(p,pop,keepers)
for j = 1:keepers
    winner = Roulette(p,pop);
    child(j) = winner;
end

CROSSOVER.m

function child = Crossover(p,nGene,nXover,pop, keepers, child)
% Create a crossover matrix
x_spot = sort(round(1+(nGene-1)*rand(1,nXover)));
% Eliminates any of the crossover spots that were generated more than once
for j = nXover:-1:2
    if x_spot(j) == x_spot(j-1)
        x_spot(j) = []; 
    end
end
% Produce the resulting offspring
nXover = length(x_spot);
counter = 1;
m = 1;
for y = (keepers+1):pop
    rents(1) = Roulette(p,pop);
    rents(2) = Roulette(p,pop);
    while rents(1).genes == rents(2).genes
        rents(2) = Roulette(p,pop);
    end
    for k = 1:(nXover+1)
if k == (nXover+1)
    r = nGene;
else
    r = x_spot(k);
end

for n = m:r
    if rem(counter,2) == 0
        offspring(n) = rents(1).genes(n);
    else
        offspring(n) = rents(2).genes(n);
    end
end

if k == (nXover+1)
    m = [];
else
    m = x_spot(k) + 1;
end

end

counter = counter + 1;
end

child(y).genes = offspring;
end

MUTATION.m

% As the subroutine name indicates, this subroutine mutates genes of % the individuals in the population based on random selection and the % mutation probability set by the user.

function child = Mutation(child,pop,nGene,mutt);

for i = 1:pop
    for j = 1:nGene
        mutate = rand;
        if mutate <= (mutt/100)
            if child(i).genes(j) == 1
                child(i).genes(j) = 0;
            else
                child(i).genes(j) = 1;
            end
        end
    end
end

DID IT CONVERGE.m

% This subroutine checks to see if the population is converging. The % subroutine returns the mean difference of all the bridges to the main % program. In the main program, if the bridges reach a certain % convergence it will break out and return a message to the user.

function converge = Did_It_Converge(p,pop,rLen);
sum_var = zeros(rLen);
sum_diff = zeros(rLen);

% Get the sum of each variable
for i = 1:pop
    for j = 1:rLen
        sum_var(j) = sum_var(j) + p(i).variable(j);
    end
end

% Get the average of each variable
ave_var = sum_var/pop;

% Sum the differences of the individuals and the averages
for i = 1:pop
    for j = 1:rLen
        sum_diff(j) = sum_diff(j) + abs(p(i).variable(j) - ave_var(j));
    end
end

% Get the average of the distances from the mean
ave_diff = sum_diff/pop;

for k = 1:rLen
    close(k) = ave_diff(k)/ave_var(k);
end

converge = mean(close);

WRITEGENETICDATA.m

function WriteGeneticData(p, generation, pop)

extension = ['Generation_' int2str(generation) '.txt'];

filename = fullfile('C:\','Documents and Settings','keith','Desktop','GA_MD'...
    ,'Genetic Algorithm','Data Bank',extension);

fid = fopen(filename,'w+');

% Information at the top of the text file
fprintf(fid,'%s
',**************************************************
*****************************);
fprintf(fid,' %s
','<< Genetic Information >>');
fprintf(fid, '%s%d
','Generation :',generation);
fprintf(fid, '%s
',**************************);
fprintf(fid, '%s		%s		%s		%s		%s		%s		%s
', 'Bridge', 'DeathYear', 'NewStress', 'GrowStress','Factor', 'KillStress', 'Mass Total');

for i = 1:pop
    fprintf(fid, '%d			%d			%d		%d		%d		%d
',i,p(i) .variable(l)
end
fprintf(fid, '%s', '****************************************************************************
*****************************************************************************');
fclose(fid);
APPENDIX B

CONVERGENCE OF MULTIPLE STRUCTURES
THROUGH SEVERAL GENERATIONS
Generation 1: Structure 3

Generation 1: Structure 4
Generation 1: Structure 5

Generation 1: Structure 6
Generation 1: Structure 7

Generation 1: Structure 8
Generation 1: Structure 9

Generation 1: Structure 10
Generation 1: Structure 13

Generation 1: Structure 14
Generation 1: Structure 15

Converged Structure