TOPOLOGY OPTIMIZATION OF AN ELASTIC AIRFOIL

Abstract

The objective of this study was to use the Hybrid Cellular Automaton (HCA) method, a topology optimization strategy developed at Notre Dame by Andrés Tovar, to find the strongest but lightest elastic airfoil which could be implemented in a morphing wing aircraft. In the study the internal structure of the airfoil was optimized. To accomplish this, the airfoil was modeled as a compliant mechanism with prescribed actuator loads and displacements. The HCA code was used to find the optimal internal structure for the compliant mechanism airfoil undergoing displacement at the leading and trailing edge as is often the case during aircraft takeoff and landing. Viable topologies were generated for the morphing wing. In addition to optimizing the internal structure of an airfoil, an algorithm suggested by Buhl was used in conjunction with the HCA method to optimize the location of attachment or support points. This new HCA code with the support optimization proved to be a successful method when studying compliance minimization in structures. When applied to compliant mechanisms, the new method led to an increase in displacement at the output locations. In the case of a force inverter, the new method led to an 8.31 % increase in output displacement. This improvement was echoed with the airfoil study. Leading edge displacements increased by 52.2 % and trailing edge displacements increased by 83.7 %.

David Lettieri

Undergraduate Thesis

Submitted: April 27, 2007

Table of Contents

1.0 – Introduction

1.1 - Background

Recently a renewed interest in morphing structures has developed, particularly in the aerospace field. This interest has led to new, lightweight, and more effective actuators [1]. With the ability to use these effective actuators to alter or "morph" the wing shape of an aircraft, the possibility of designing aircraft with morphing wings is becoming a more realistic possibility. Certain morphing mechanisms are already in use today. Examples of these morphing structures include the high lift systems on commercial airliners and the variable sweep wing on the Navy's F-14A Tomcat [2]. Elastic morphing is yet another method of morphing wings in which the shape of a wing with an elastic skin is changed via the movement of hydraulic or pneumatic actuators. This is the method of morphing under investigation in this study.

Wing morphing is desirable because it allows a single aircraft to undergo different types of missions [3]. For instance, a mission with such combined objectives as high altitude cruising for long periods of time and then high speed chase and attack utilizing agile movements would be a mission where wing morphing is advantageous. Since the wing shape can be altered, the different mission objectives can be accomplished, whereas without a morphing wing, the mission either would be impossible or would be inefficient because excess fuel would be consumed. However, morphing wings do not always provide a benefit. Morphing structures add weight to the aircraft which in turn necessitates more fuel for a mission. Therefore, in order to be beneficial, the fuel saved by the ability to morph must be greater than the fuel lost by the additional weight of the actuators.

In order to optimize the performance of morphing wings, wing structures must be obtained that minimize weight while maximizing strength. In this study the objective was to use computer simulations that utilized the HCA topology optimization strategy to find the strongest but lightest elastic airfoil. The airfoil was modeled as a compliant mechanism with prescribed actuator loads and displacements. In order to understand the work done in this study, first one must understand topology optimization as well as compliant mechanisms.

1.2 – Topology Optimization

Topology optimization was the method used to find the optimal internal structure for an airfoil modeled as a compliant mechanism undergoing displacement at its leading and trailing edge. Topology optimization is an iterative numerical technique that attempts to distribute material in a finite volume in such a way that optimizes a parameter of interest. There are various forms of topology optimization, each of which uses a different numerical method. The Hybrid Cellular Automaton (HCA) method, which was the optimization scheme used in this study, was developed at the University of Notre Dame [4]. This method discretizes a given design domain into elements and alters certain design parameters. As is most often the case, in this study the relative density within each cell was the design parameter, which was varied between approximately 0 and 1.

Different material models can be used to model the design variables. The HCA method uses the Solid Isotropic Material with Penalization, or SIMP, approach [5] [6]. In this method the material properties are assumed constant within each element. The elastic modulus of each cell was calculated using a base modulus multiplied by the

relative density raised to a penalization power. Equations 1 and 2 illustrate how the elastic modulus, E_i, and density, ρ_i, for each element are calculated from the relative density. In these equations, x_i represents the relative density of the i^{th} element.

$$E_i(x_i) = x_i^p E_o \tag{1}$$

$$\rho_i(x_i) = x_i \rho_o \tag{2}$$

The penalization power, p, was equal to 3 and was used to drive the relative density to 0 or 1. The power of 3 was necessary to drive elements to full density or no density in order to eliminate intermediate densities from the design domain since intermediate densities have no real meaning. Minimum and maximum extremes of $1X10^{-3}$ and 0.999 were set for the relative density so that the matrix that stored the relative density did not become singular. These limits also allowed a given element to be reintroduced in a future iteration. The HCA method will be developed more thoroughly in the methodology section.

Topology optimization seeks to optimize certain quantities of interest. In this study two different quantities were optimized. When optimizing the internal structure of a stiff structure, strain energy density, *SED*, was minimized. When optimizing the internal structure of a compliant mechanism, mutual potential energy, *MPE*, was minimized. These quantities used were based on the formulae developed below.

Strain energy, often notated as U, is the energy stored in a structure resulting from elastic deformation. This amount of energy is equal to the work necessary to perform this deformation. If the structure behaves linearly, the force is proportional to the displacement and the work is calculated as shown below. Note that this equation holds for one dimensional axial strain where the force and the displacement are aligned.

$$U = W = \frac{1}{2} * F * d \tag{3}$$

The distance term, *d*, in equation 3 is the deflection due to deformation and is often noted as δ. In turn, this term can be replaced by the strain of the element of interest, ε. Note that in equation 4, *l* is the length of the element. In this investigation square elements were used so that the length of the element was the same as the height of the element. Also, it is important to note that this study was a two-dimensional study. This means that airfoil cross-sections were studied as opposed to three dimensional wings.

$$\delta = \varepsilon * l \tag{4}$$

The force term in equation 3 can be replaced by definition with equation 5 below, where σ stands for stress.

$$F = \sigma * l^2 \tag{5}$$

The total strain energy for a discretized domain is defined below in equation 6. Note that N_e is the number of elements in the design domain, U_i is the strain energy density, *SED*, and v_i is the volume of element *i*. Again, these were two dimensional elements, so in reality there is no depth to the volume. However, for the derivation of strain energy, it was assumed that cube elements were being modeled with a depth of distance *l*. This extra factor of *l* has no significance in the use of strain energy density for two dimensional analyses. Therefore, *SED* can be defined as the strain energy per unit volume.

$$U = \sum_{i=1}^{Ne} U_i v_i \tag{6}$$

Finally, it is useful to note that by definition of the elastic modulus, E, equation 7 is true.

$$\sigma = \varepsilon * E \tag{7}$$

When equations 3 through 7 are combined, the result is the strain energy density formula shown below in equation 8.

$$U = \sum (U_i * l^3) = \sum (\frac{1}{2} * \frac{F_i * d_i}{l^3} * l^3) = \frac{1}{2} \sum \left(\sigma_i * l^2 * \varepsilon_i * l \right)$$

$$= \frac{1}{2} \sum \left(\sigma_i * \varepsilon_i * l^3 \right) = \frac{1}{2} \sum \left(\varepsilon_i^2 * E_i * l^3 \right) \quad (8)$$

By examining equation 8, it is obvious that to make a stiff structure that has minimal deformation, strain, and therefore *U* or *SED*, must be minimized. Also *SED* is a function of E_i, which for each element is a function of x_i as defined in equation 1. Therefore, varying x_i will alter the value of E_i which will lead to changes in the compliance and rigidity of the structure. Finally, note that this equation was derived here for axial strain in one direction with the stress and strain being colinear. This equation can be expanded for two or three dimensional strain. The more complicated formula has the same tendencies and was used in the finite element analysis.

Mutual potential energy is similar to strain energy as shown in equation 9 [7].

$$MPE = \sum \left(\sigma_{i,output} * \varepsilon_{i,input} * l^3 \right) = \sum \left(\varepsilon_{i,output} * \varepsilon_{i,input} * E_i * l^3 \right) \quad (9)$$

The first difference is that by definition, *MPE* does not contain the factor of one-half by definition [7]. The second difference is more subtle and requires knowledge of load cases. A load case is a set of loads that occur simultaneously. When modeling compliant mechanisms, as is described in the compliant mechanism section below, an input force due to an actuator coupled with an external spring is modeled to obtain the appropriate input displacement. Similarly, a dummy output force and external output spring stiffness is modeled to obtain the correct output displacement. The input and output loads are part of two different load cases. When defining mutual potential energy, the stress term in

equation 8 is a result of the dummy output load, while the strain term is a result of the input load [8]. The effect is to relate the input actuator loads and the dummy output loads. When studying compliant mechanisms, mutual potential energy is minimized. The initial setup of a compliant mechanism, including the dummy load and external spring at the output location, causes the strain and thus the deflection at the output location to be maximized in order to get the highest flexibility and displacement at the point or points of interest.

1.3 - Compliant Mechanism

A compliant mechanism is a mechanism in which displacements occur due to applied loads and the elastic nature of the mechanism. Compliant mechanisms are made from one piece and do not require moving parts for their motility. Figure 1 is an example of pliers that are a compliant mechanism.

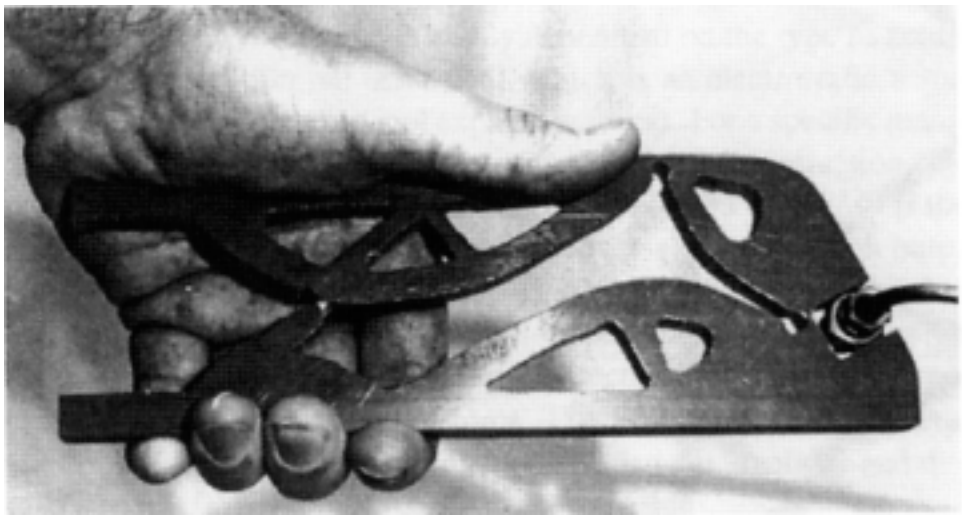

Figure 1: Compliant mechanism pliers [9].

The force supplied by a person's hand on the pliers is considered an actuator force. An approach mentioned by Sigmund [8] can be used to constrain certain portions of the design domain to have certain displacements rather than defining the displacements explicitly. This is done to create compliant mechanisms with displacements where

desired. For example, when modeling the compliant mechanism pliers in figure 9, Sigmund's method would be used to prescribe displacements where the pliers pinch. Sigmund's method involves specifying a dummy load at any point desired, along with an external spring. The displacement of this point can be found using equation 10 below. Since a finite element analysis will be used to calculate displacements, points will be referred to as nodes as is convention in finite element analyses. Note that in the finite element analysis all three of the variables below are matrices with values for every node.

$$u = \frac{F}{k} \tag{10}$$

In equation 10, k is the stiffness of the node of interest. This stiffness consists of more than just the external spring that has been modeled as attached to this node. In addition to this external stiffness, each node has a stiffness due to the elastic nature of the material. This inherent stiffness is found using a global stiffness matrix in the finite element code and relates all the stiffnesses of all the nodes. The stiffness, *k*, in equation 10 is a sum of the stiffness of the node calculated by the finite element analysis as well as the external stiffness added at that node. The inherenet stiffness due to the elastic nature of the material cannot be known ahead of time since it is a function of the mass of the element that the node belongs to, and the mass varies from iteration to iteration in the topology optimization scheme. As a result, even though the input force is defined, and an external spring is added, the displacement at this node is not known due to the variable nature of the inherent stiffness of the material. However, the external stiffness is comparable in magnitude to the inherent stiffness and has an effect on the displacement. Using this method, one can obtain displacements fairly close to desired at nodes where external forces and springs are assigned.

This method of adding an external spring can be applied to mechanisms of any sort. In this study, the goal was to apply this method to an airfoil. By using external springs and dummy forces at the leading and trailing edge, the displacement at these locations can be controlled. The shape of the airfoil can be manipulated to increase the camber of the airfoil much the same way leading and trailing edge slats and flaps operate on commercial airliners with conventional morphing technology. A NACA 0012 airfoil can be morphed into an airfoil with camber. However, it was not the aim of this study to morph the airfoil into a shape optimal for landing. Rather, the aim was to develop a method of morphing a wing so that such an optimization could be performed in conjunction with a computational fluid dynamics modeling of the wing. This will be discussed further in the discussion section.

1.4 – Support Optimization

When using topology optimization techniques, initial conditions must be known prior to the optimization. One such initial condition is the attachment point or support location. Fixed regions, or regions that are constrained such that they do not undergo displacement, are traditionally arbitrarily chosen before optimization. Combining a method proposed by Buhl [10] with the HCA method, these fixed regions were not chosen arbitrarily but instead were also optimized. This new feature increased the utility of the HCA method.

As Buhl suggested, the SIMP approach was used with "relative stiffness", q, as the design parameter in support optimization much the same way that relative density was the design parameter optimized for structural optimization. Nodes that were in the attachment optimization region had springs added to them to model either free or fixed

nodes. The stiffness of the spring was varied from a very high stiffness to simulate a fixed node to a low stiffness to simulate a free node. Again penalization was used to drive intermediate stiffnesses to the minimum or maximum value of $1X10^{-4}$ and 0.999 respectively, via a penalization power. However, in the support optimization the penalization power was set to 4 rather than 3. The change in the value of the penalization as well as the minimum relative stiffness is explained in the support optimization section of the methodology section. Equation 11 below shows how the SIMP approach was used in the support optimization. Support optimization will be discussed further in the methodology section.

$$k_i(q_i) = q_i^p k_o \tag{11}$$

2.0 – Methodology

2.1 – The HCA method

In order to understand the methodology used in this investigation, a good understanding of the HCA method is necessary. For a more complete explanation of this method see Tovar's paper on the HCA method [4]. Below is an abridged explanation of the HCA method. True cellular automaton (CA) methods involve only local information. HCA is referred to as a hybrid method because it involves using global information to make decisions with a local control strategy. The design domain is discretized into a mesh of elements for global analysis and a mesh of cells for local analysis. In this study these two meshes were the same; however, they do not have to be. Information can be mapped from one mesh to another if they differ. In this paper, since elements and cells are equivalent, they will be referred to as cells and elements interchangeably. Also in this explanation of the HCA method, the parameter to be optimized will be *SED*. The algorithm works in exactly the same fashion if one was optimizing to achieve a target *MPE* or other parameter.

This method has six main iterative steps. The first step in HCA is to define a design domain, loads, supports, initial design, and material properties. This initial design domain is discretized into elements for global and local analysis of field variables. A typical initial design domain is shown below in figure 2. The initial design domain includes the entire area for structural optimization to occur. The figure shows the design domain for a force inverter compliant mechanism that was used in this study as a simple example and foundation for more complex mechanisms, such as the compliant mechanism airfoil.

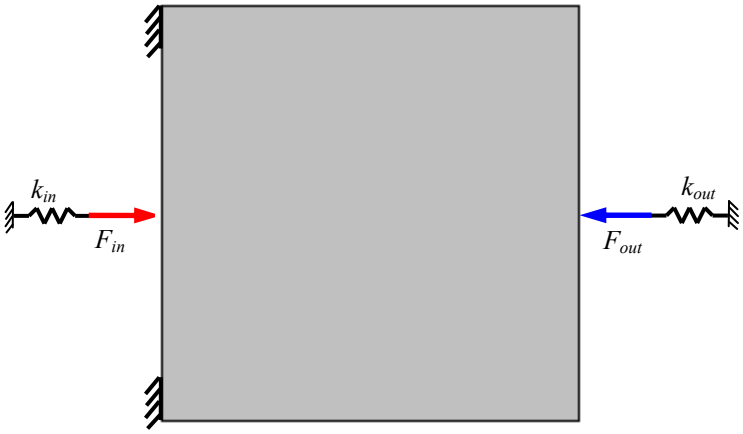

Figure 2: Traditional force inverter design domain.

In this example the initial design domain is the gray rectangle. In figure 2 the red and blue arrows represent the input and output loads, respectively. Throughout this paper, red arrows will represent input loads, blue arrows will represent output loads, and hash marks will represent fixed nodes. Zones of hashed marks encompassed by a rectangular perimeter will represent regions of support optimization.

The red and blue loads in figure 2 above are part of two separate load cases used in the initial loading definition. The external springs and fixed boundary conditions are also part of the initial loading definition. Color is used in the HCA method to represent graphically the relative density in the design domain. Areas of no material or zero density will be represented by white, while areas of full material or full density will be represented by black. Intermediate densities will be shaded gray with the intensity proportional to the intermediate density. The initial design domain in figure 2 has no initial geometry. The HCA algorithm will distribute mass throughout the region to optimize the parameter of interest beginning with an arbitrary initial relative density. However, an initial geometry can be programmed. This feature was used with the airfoil optimization. A NACA 0012 airfoil initial geometry inside of a rectangle was the initial

geometry as shown in figure 3. It is easier to have a rectangular design domain with an internal initial geometry than to have a non-rectangular design domain. Non-rectangular design domains prove more complicated when performing a finite element analysis.

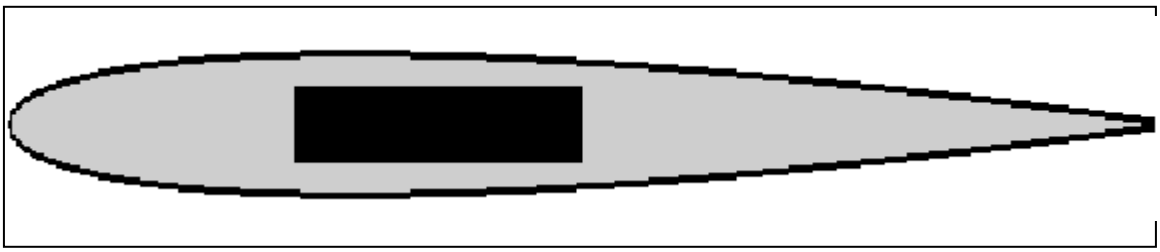

Figure 3: Example of a design domain with an initial geometry (NACA 0012 airfoil).

The goal of the HCA method is to alter the relative density in each element to optimize a certain characteristic. However, to maintain a certain initial geometry, passive elements can be assigned. These elements either can be forced to have full material or no material and remain constant for all iterations. In figure 3 the black skin and wing box are areas that have been constrained to always have material ($x_i = 0.999$). The white area outside of the skin but inside of the rectangle that defines the design domain has been constrained to always be void of material ($x_i = 0.001$).

The material properties for each element are considered constant throughout the element. The two main material properties of interest are the elastic modulus and Poisson's ratio for the material. These values affect the stress and strain calculated in the finite element analysis discussed below and are dependent on the relative density of each cell. In this manner the relative density affects the deflections in the design domain.

The second step is to evaluate the stress and strain for each element via a global structural analysis, which is accomplished through a finite element analysis. This analysis calculates the stress and strain for every node within the design domain. This is the global information portion of the HCA algorithm. Using the stress and strain

information from each node, the *SED* for each element was calculated. The *SED* for all the elements was averaged to be used as a target *SED*.

The third step is to evaluate the *SED* in a CA neighborhood. In HCA a local neighborhood of cells is designated for the purpose of applying a local design rule. This is the local information portion of the HCA algorithm. The neighborhood could have several definitions, but for this study the neighborhood consisted of the four adjacent cells to the cell of interest as well as the cell of interest. Figure 4 below shows the Von Neumann (N =4) neighborhood used in this analysis. Other neighborhoods are the Moore (N = 8), MvonN (N = 12), and the extended Moore (N = 24) [4]. Neighborhoods are necessary to prevent certain numerical anomalies that may arise. For instance, one such anomaly is checkerboarding. This phenomenon, which occurs to minimize compliance while using little mass, is a checkerboard type topology with alternating cells of full and no density. By creating neighborhoods, these phenomena are avoided by the averaging over the neighborhood. Larger neighborhoods generally work better but require more computational time.

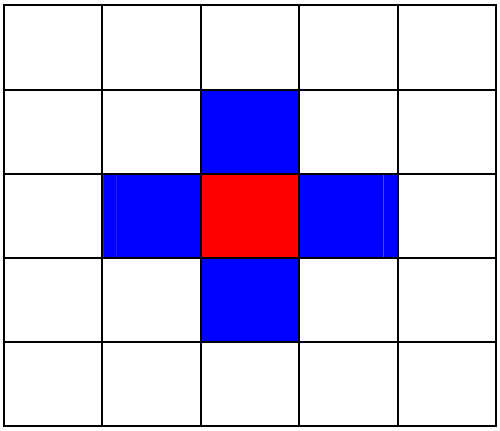

Figure 4: Von Neumann (N=4) neighborhood. The red element is the cell of interest. The blue cells are the neighborhood.

Once the neighborhood was determined, the *SED* was averaged over the neighborhood.

The fourth step in the HCA method is to compare the average *SED* of the cell of interest to the target *SED*. The goal of HCA was to drive the difference between the *SED* average and the target *SED* to zero. Since the target *SED* was the average *SED* of all the cells, by driving the local average to the global average, the *SED* is being evenly distributed throughout all the cells. The consequence is a minimization of the *SED* and deflection in the structure.

The *SED* average for each neighborhood was driven to the target by changing the relative density of the cell of interest using a control strategy. Step five of the HCA method could implement many different control strategies; in this study PID was the control strategy used to change the relative density of the cell as a function of the difference between the *SED* average for the neighborhood and the target. If a particular neighborhood had a *SED* average below the target, the change in relative density would be negative. The reason for this is that if the average *SED* was lower than the target, the neighborhood would be deflecting less than the target. In order to deflect more, mass would have to be removed from the cell of interest. Similarly, if the average *SED* for the neighborhood was above the target, the change in relative density would be positive. Once calculated, the change in relative density of the cell was then added to the previous relative density of the cell. This was done for all cells simultaneously.

It is important to note that when calculating the new relative density for each cell, the total mass of the design domain was constrained to remain constant. The mass ratio or mass fraction was defined as the sum of all the relative densities for all cells divided by the total number of cells. Therefore, if the design domain began as having a 20 % mass ratio or global mass fraction, this 20 % mass ratio was maintained for every

iteration. As a result, mass was transferred from one point to another in order to optimize either *SED* or *MPE*, depending on whether a stiff structure or compliant mechanism was being analyzed. Also, move limits were placed on the change in relative density for each cell so that dramatic changes in relative density for a given cell could not occur in any iteration. This helped to prevent instabilities.

Once completed, the final step in the HCA method was to check to see if there had been a significant change in the overall relative density change from the previous iteration. If there had been a significant change, the algorithm began again with a finite element analysis. If convergence had been achieved, the final topology had been obtained, and the algorithm ended. Figure 5 summarizes the HCA algorithm.

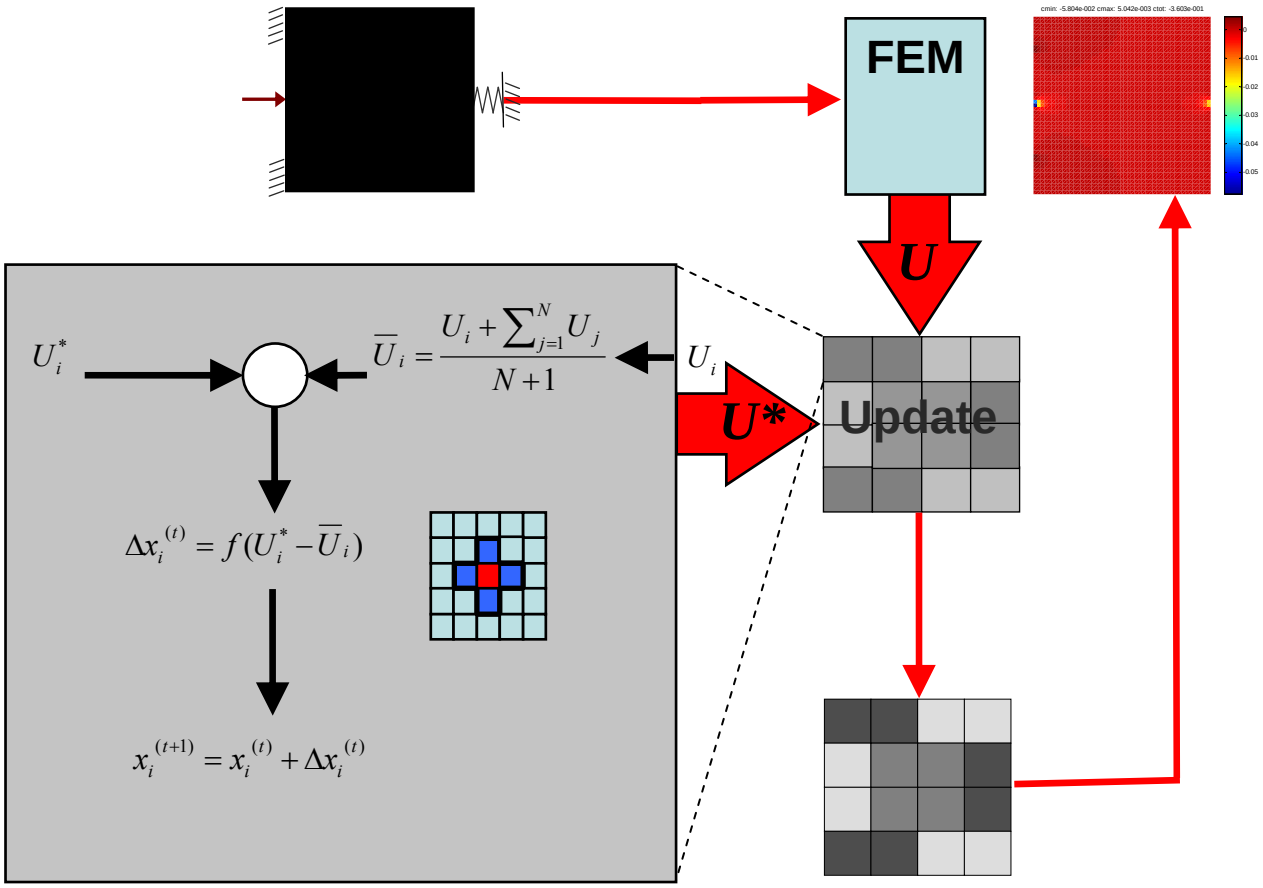

Figure 5: Flow chart of HCA method and its iterative nature [9].

2.2 – Support optimization

Support optimization allows for one less initial condition to be specified at the beginning of the optimization process. Instead of arbitrarily choosing fixed regions to act as supports, regions can be assigned where an algorithm will calculate the best location for supports within that region. A location will be considered the best region for a support if it makes the design stiffer. The change in the initial design domain definition for the force inverter example, as compared to figure 2, is shown below in figure 6.

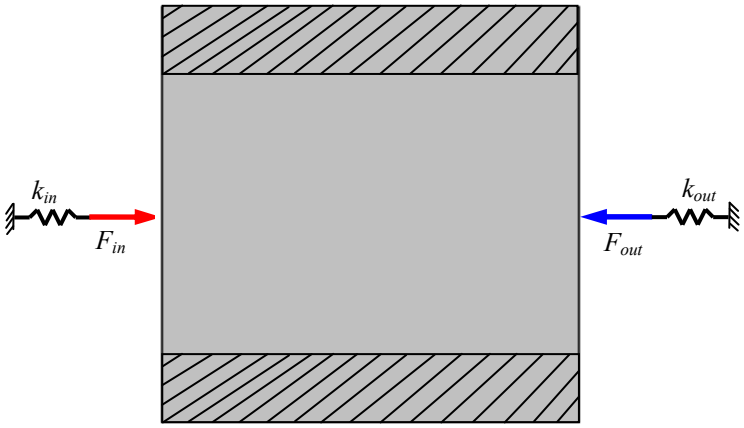

Figure 6: Initial design domain for a force inverter undergoing support optimization.

Within the regions for support optimization, an optimization scheme similar to that for a stiff structure occurred. However, in regions of support optimization, rather than simply optimizing the internal structure by varying a relative density in each cell, a relative stiffness in each cell was also varied. Both design parameters were optimized to make the structure stiffer. This was accomplished by minimizing the *SED*. A model of the cells in the support optimization zone is shown in figure 7.

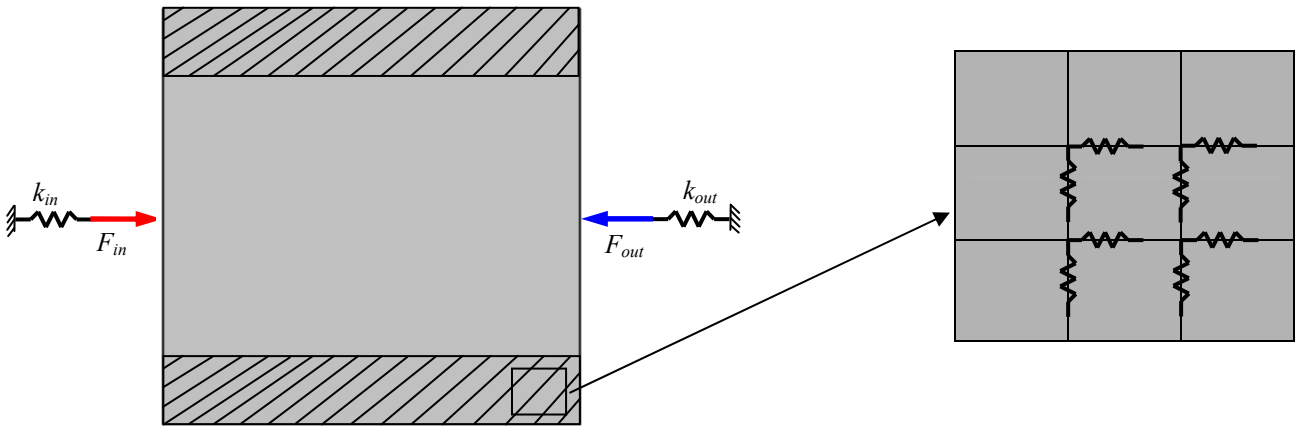

Figure 7: A typical cell in the support optimization region.

Figure 7 illustrates that within the support optimization region, springs were attached in both directions to the four nodes around a given cell. By varying the stiffness on these springs from nearly 0 to very large, a cell could be considered free or fixed. The stiffness of these springs was determined using a relative stiffness for each cell and the SIMP approach as defined in equation 11.

Each spring for a given cell had a base stiffness. The 8 springs added to the i^{th} cell had a stiffness that was the product of the relative stiffness of that cell raised to a penalization power multiplied by a base stiffness. The base stiffness was $1X10^{10}$ so that if the relative stiffness was near 1, the stiffness of that cell would be nearly $1X10^{10}$. A stiffness that large essentially created a fixed node. The minimum value for the relative stiffness was set as $1X10^{-4}$ so that when raised to a penalization power of 4 and multiplied by the base stiffness, the resulting stiffness for that spring was much less than 1, which produced the effect of a free node. The reason for using a penalization power of 4 rather than 3 as in the material optimization and for decreasing the minimum relative stiffness bound from $1X10^{-3}$ to $1X10^{-4}$ is that the stiffness calculated using equation 11

which results from the minimum relative density is much smaller and much less than 1 than if the former had been used. The result is to ensure a "free node" condition when the relative stiffness value is at its minimum value.

The process of distributing the relative stiffness within the cells undergoing support optimization was similar to the normal HCA method. The regions of support optimization underwent the HCA method but with 2 design parameters, x_i and q_i, being optimized. The entire design domain was discretized into a mesh as is normally the case for the HCA method. Regardless of whether a stiff structure or a compliant mechanism was being optimized, the same method was used for the support optimization. The *SED* for the entire design domain was calculated and averaged for a target *SED*. If a compliant mechanism was being optimized, the *MPE* for the entire design domain also was calculated and averaged for a target *MPE* concurrently. In the regions of the design domain that were not part of the support optimization, optimization occurred normally as would be the case without support optimization. Therefore, if a compliant mechanism was being optimized, the *MPE* was calculated and optimized. If a stiff structure was being optimized, the *SED* was calculated and optimized. Within the regions of support optimization, whether optimizing the structure for a compliant mechanism or a stiff structure, the *SED* average was calculated for each neighborhood. Structural optimization to determine the relative density in the support optimization regions still occurred as normal. However, in addition to structural optimization, support optimization occurred. If the *SED* average for the neighborhood was lower than the target, the relative stiffness for that cell of interest was decreased to make that cell less stiff. If the *SED* average for the neighborhood was higher than the target, the relative

stiffness for that cell of interest was increased to make the cell stiff. The stiffness for each spring attached to that cell was calculated using the SIMP approach as described in equation 11.

Again, a global stiffness fraction, analogous to a global mass fraction, was maintained for each iteration as was done for the relative density in the normal HCA method. If the region of support optimization began with a 5 % of the region having full relative spring stiffness, this ratio was maintained for every iteration. Move limits also were placed on the change in relative stiffness.

Several important issues arise when optimizing support location. First, one might consider including the entire design domain for support optimization. However, if this was done, supports would be placed in areas of high stress and strain. In the case of compliance minimization, rather than creating an optimal structure, supports would be placed where structure would normally be placed as a method of reducing the strain in the structure. This would not make any sense from a design point of view. To resolve these problems for both compliance minimization and compliant mechanism, regions for support optimization had to be constrained to portions of the design domain.

Second, one might question the method of minimizing compliance in a compliant mechanism. As was stated earlier, in compliant mechanism synthesis, the goal is to obtain the largest displacement at the output location. It seems contradictory to minimize compliance in a compliant mechanism through the support optimization if the goal is to obtain maximum deflection at certain portions of the design domain. To understand why compliance in a compliant mechanism is minimized at support locations, a sponge may be considered. A sponge is highly compliant meaning that forces on a sponge result in

large deflections. However, a sponge would not be a very efficient compliant mechanism because it is too flexible. Compliant mechanisms need to combine flexibility for movement with rigidity so forces can be transmitted. The compliance minimization only occurs in support optimization regions. This results in supports that make the support structure as stiff as possible. The material distribution due to the initial loading remains geared towards making the structure flexible.

The third area to mention briefly is the linked nature of the eight springs related to a given cell. Figure 7 shows how a cell in the support region is modeled. A cell in the support region has 8 springs constraining the motion of each of its nodes in both the horizontal and vertical directions. All eight of these springs are linked by the fact that they have the same penalization power, the same relative stiffness from that cell, and the same base stiffness. Therefore, the stiffness of all eight of these cells are the same. This is an advantageous phenomenon because it prevents regions from being fixed in only one direction. Extreme structures could result from this form of a fixed region [10]. By linking all eight spring stiffness, regions fixed in both the horizontal and vertical direction are found.

3.0 – Examples Studied

3.1 – Airfoil Optimization with Original HCA Method

The first configuration examined was the NACA 0012 airfoil with the original HCA algorithm. A rectangular design domain was constructed that was 40 by 240 elements. A skin was defined as always active, or full density; therefore there was an initial geometry. Also, a wing box was defined as always active. This region modeled a wing box that goes through a wing giving structural support as well as storage space for components or fuel. The region outside of the wing skin but within the rectangular design domain was defined as passive, or no density. This area appears as white below in figure 8, which shows the initial conditions for the NACA 0012 airfoil.

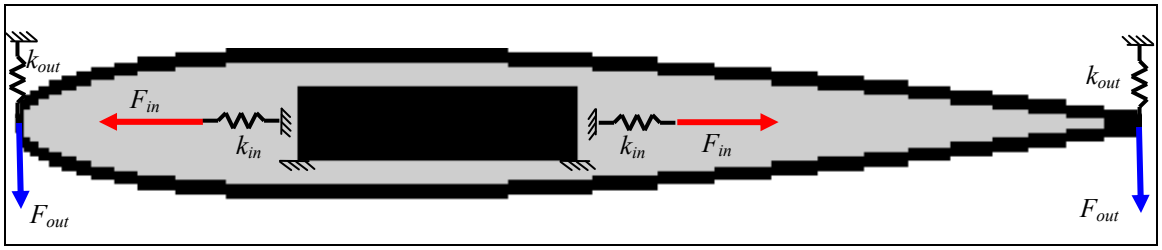

Figure 8: NACA 0012 airfoil initial design domain and initial conditions.

Several factors were varied in this example. These factors were thickness of the skin, mass fraction, support location, and actuator load location. Also, the same initial design was investigated with a vertical force applied to roughly simulate the atmosphere.

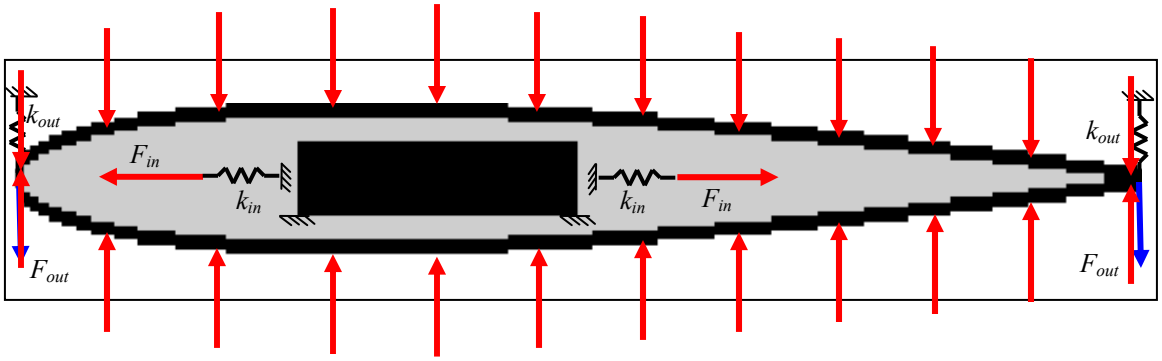

Figure 9: NACA 0012 airfoil with unit force per area.

3.2 – Support Optimization with Compliance Minimization

Before examining an airfoil with the support optimization algorithm, various simpler studies were conducted to validate that the algorithm combining support optimization with compliance minimization was working correctly. The first validation study involved stiff structures. In this optimization the goal was to minimize compliance to make a stiff structure. First, a rectangular design domain with a force at the center of the top of the domain was analyzed. The bottom surface of the design domain one row of elements thick was set as the support optimization region. This design domain can be seen in figure 10 below.

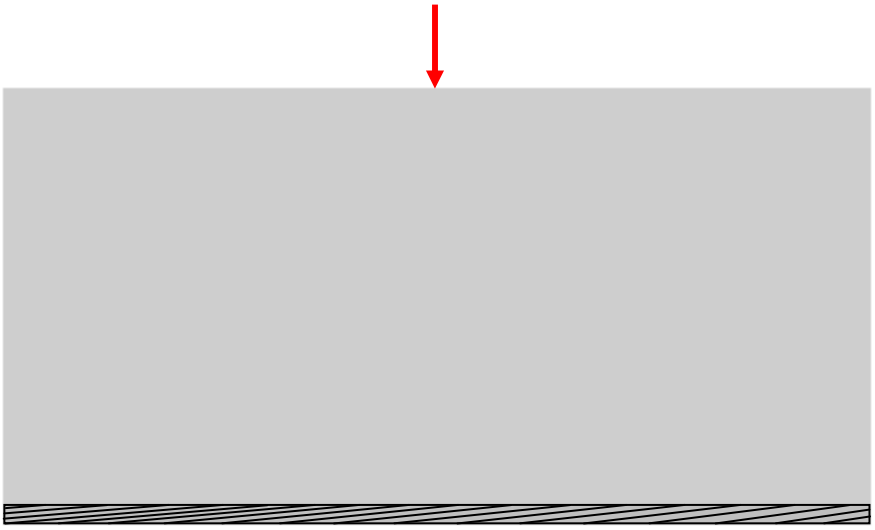

Figure 10: 40 X 80 rectangular design domain with load at top center and optimization region at bottom. The initial mass fraction was 0.2.

The second structure analyzed was a rectangular design domain with support optimization regions along the sides of the design domain. The load was centered in the design domain. This is illustrated in figure 11.

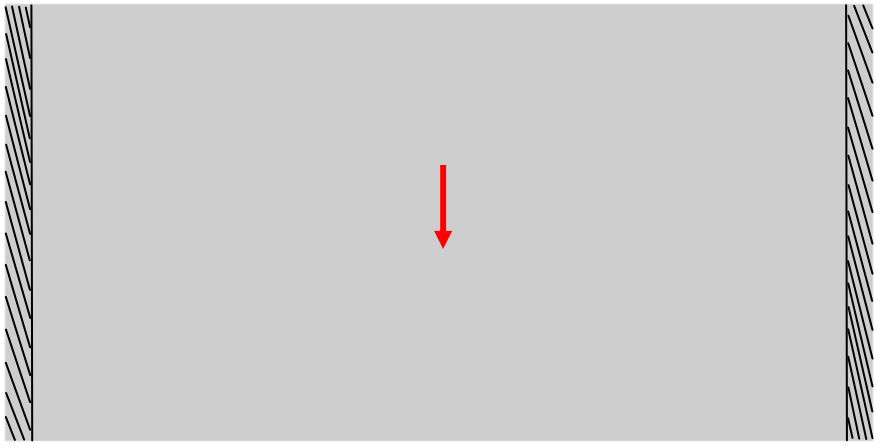

Figure 11: 20 X 40 rectangular design domain with load centered horizontally and vertically and optimization region on sides. The initial mass fraction was 0.2.

The final structure analyzed was a rectangular design domain with support optimization regions along one side. Again the load was applied in the center at the top of the design domain as was done in the first structure.

Figure 12: 20 X 40 rectangular design domain with load at top center and optimization region on left side. The initial mass fraction was 0.2.

3.3 – Support Optimization with Simple Compliant Mechanisms

The next step after verifying that the support optimization code worked with compliance minimization was to verify that the algorithm resulted in improved results for compliant mechanisms. Better results would mean more displacement at output locations. To

examine whether or not better results would be achieved, a force inverter was analyzed. The initial conditions for the design domain are shown below in figure 13.

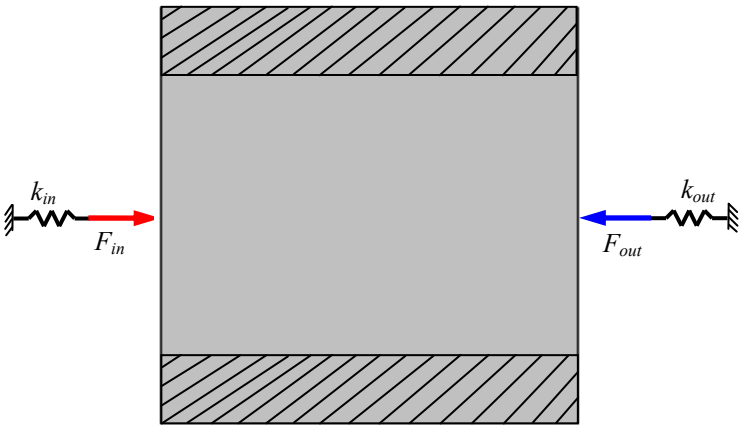

Figure 13: 40 X 40 force inverter design domain with a mass fraction of 0.2.

The input and output loads were both 1.0. The input external spring stiffness was 1.25, and the output spring stiffness was 0.025. All loads in this study were of magnitude 1. Stiffness, loads, and the elastic modulus were all in the order of magnitude of 1. The reason for this is that it makes modeling less complicated with regard to maintaining consistent units and correct magnitudes. As long as the analysis remains linear there is no ill effect since these quantities are all scalable. A negative consequence is that this keeps the problem under investigation an academic problem. However, simple scaling of these quantities so that they did have units would not alter the topology generated.

For the force inverter, the top and bottom 10 % of the design domain were the support optimization regions. The global stiffness fraction was 5 %, and the global mass fraction was 20 %. A global stiffness fraction of 5 % means 5 % of the support location region would be assigned as locations for attachment or fixed nodes.

3.4 – Support Optimization with NACA 0012 Airfoil

The final configuration studied was the NACA 0012 airfoil with support optimization instead of a wing box. The initial design domain, loading, and support

optimization region can be seen in figure 13 below. This design domain was also 40 by 240 elements as in the first case and had a mass fraction of 0.238 as well as a global stiffness fraction of 5 %. The gray region of the initial design had a relative density of 0.2. Again, the skin of the airfoil drives the mass fraction to 0.238. However, this does cause any significant changes.

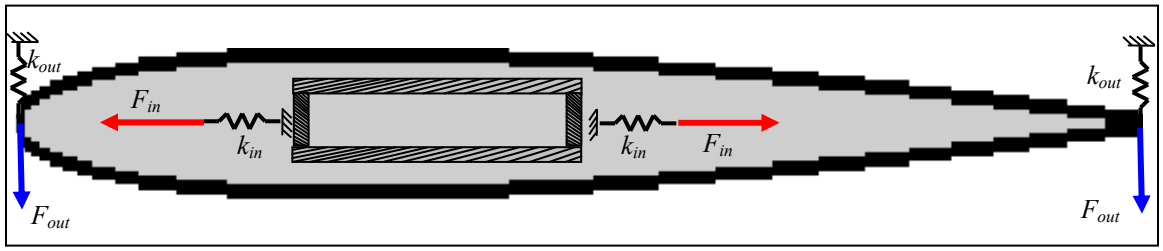

Figure 14: 40 X 240 NACA 0012 design domain with a mass fraction of 0.238. Downward displacements are being forced at the leading and trailing edges.

4.0 – Results

4.1 – Airfoil Optimization with Original HCA Method

Case I: Thick Skin with Supports at Bottom Corners of Wing Box

The first initial design studied was the airfoil as pictured exactly in figure 8. The mass fraction for this design was 0.318. This corresponded to an initial relative density of 0.2 for the gray region, 0.999 for the black region, and 0.001 for the white region. Since there was a large amount of full density material, the initial mass fraction was larger than 0.2 as was the goal for the structure. This small deviation was not significant. This first case will be the comparison for cases II through VI. The topology generated is shown below in figure 15.

Figure 15: Airfoil configuration of thick skin, mass fraction = 0.318, supports at bottom corners of wing box (green points), and actuator loads at red point points.

For the remainder of the results section a color convention will be used for ease of interpreting the results. The convention will be the follwing: red points represent the location of input loads as shown in figure 8, and green points represent fixed nodes.

This topology seems reasonable. When loaded at the point closer to the leading edge with a force pointing towards the leading edge, as indicated in figure 8, the force pushes on the structure present which is anchored to the fixed node. As a result the structure rotates and pushes down on the leading edge causing a downward deflection. The diagonal member that formed at the leading edge is there because of the external spring modeled at the leading edge. This spring absorbs some of the energy of the deflection causing the very leading edge of the airfoil to resist downward deformation.

Consequently, the diagonal member helps to force the leading edge downward by distributing some of the resistance due to this spring to an area with more downward force closer to the actuator load.

The actuator load closer to the trailing edge is pushing on structure that is forming that will force the trailing edge downward. This region is not clearly defined. It is important to note that this topology did not converge. After running a defined maximum of 100 iterations, there had been no convergence. All airfoils in this report did not converge but instead ran for the maximum of 100 iterations. This is partially the reason for the lack of black material in the structure at the rear of the airfoil. Another part of the reason is that a mass fraction of 0.318 was maintained. More stress and deformation occurred at the leading edge due to the proximity of the leading edge actuator to the leading edge compared to the trailing edge actuator to the trailing edge. As a result, more material was place at the leading edge, and the mass fraction of 0.318 was not sufficient to place more material at the rear of the airfoil.

Overall, the topology generated seems reasonable when loaded. For all airfoil cases being studied in the results section of the report, deflections reported for the leading and trailing edge will be the the average of the first 3 nodes deflections. This was done to get a more accurate deformation for comparison. The deformation at the leading edge of this case was 0.268. The deformation at the trailing edge of this case was -0.0356. The reason that this deflection was negative or upward was again because of the external spring. Not enough support material was distributed at the trailing edge to transmit the actuator load to the trailing edge.

Note that the displacements listed do not have units. This is again because the forces, stiffness, and elastic modulus did not have units. Also, the design domain itself had no prescribed units. It was merely 40 elements by 240 elements. This work, as long as displacements remain linear, can be scaled for different systems of units resulting in the same topology. This will be the case for all results in this report.

Case II: Thin Skin with Supports at Bottom Corners of Wing Box

The resulting topology for thin skin with supports at the bottom corners of the wing box is shown below in figure 16.

Figure 16: Airfoil configuration of thin skin, mass fraction = 0.229, supports at bottom corners of wing box (green points), and actuator loads at red point points.

This topology does not make sense. As in case I, the structure at the leading edge of the airfoil will push the leading edge downward when pushed on by the leading edge actuator force. However, no structure formed near the trailing edge. It is possible that the analysis was skewed because of the fact that the skin is only 1 element thick. Skin elements have integer values. When changing from one integer to another in the vertical direction, for instance from element 19 to 20, there is a step change. The result is two elements that are only connected by the corner they share. This type of configuration can be problematic in a finite element analysis. As a result, the remainder of the airfoil studies were conducted with skin 3 elements thick to ensure that elements in the skin were always touching the face of the elements of skin to their immediate left and right. A potential drawback of the 3 element thick skin is that it is relatively thick. However,

because better topologies were generated with the thicker skin, it was used in all further studies.

Case III: Thick Skin with Supports at Bottom Corners of Wing Box but Increased Mass Fraction

The third topology generated was for an initial design domain exactly as pictured in figure 8, except that the initial relative density of the gray elements was changed from 0.2 to 0.3. This resulted in an overall mass fraction of 0.352 when the skin and wing box were considered. Figure 17 shows the topology that was generated.

Figure 17: Airfoil configuration of thick skin, mass fraction = 0.352, supports at bottom corners of wing box (green points), and actuator loads at red point points.

This topology is similar to the topology generated in case I. However, the structure at the trailing edge of the airfoil is more defined because more mass was available. As a result, a larger deflection at the trailing edge occurred. Also, the region at the top of the leading edge is more defined and is closer to a converged answer. The result of these 2 phenomena is more deflection at both the leading and trailing edges. The deflections at the leading and trailing edges were 0.368 and 0.1187, respectively. This topology generated was the best topology of those generated with the normal HCA algorithm because it had the largest leading edge deflection without considering the topology generated with a unit "pressure" force and the best shaped topology.

Case IV: Thick Skin with Altered Support Location

Figure 18 is the topology generated when the fixed nodes were changed from the bottom corners of the wing box to the top corners. This topology is very similar to the

topology for case I, which was virtually the same case except with the supports in the lower two corners rather than the upper two corners. Slight differences in the extent to which the solution converged make the structure appear slightly different. However, since the wing box was modeled as a solid block, if one corner is fixed, the entire box is essentially fixed. As a result, changing the two fixed corners from the bottom to the top had little effect. The structure attached to the wing box in a similar manner to case I, and because the solid wing box is quite stiff, similar displacements occurred. The exact same deflections at the leading and trailing edges occurred out to three significant figures for both cases. The deflections at the leading and trailing edges were 0.268 and -0.0356, respectively.

Figure 18: Airfoil configuration of thick skin, mass fraction = 0.318, supports at top corners of wing box (green points), and actuator loads at red point points.

Case V: Thick Skin with Supports at Bottom Corners of Wing Box and Altered Actuator Load Location

The next topology generated using the normal HCA algorithm is shown below in figure 19.

Figure 19: Airfoil configuration of thick skin, mass fraction = 0.318, supports at bottom corners of wing box (green points), and actuator loads at red point points.

This topology was the result of an initial design as shown in figure 8 except that the actuator loads were moved out from the center toward the leading and trailing edges 10

units each. As a result the structure that was formed by the algorithm was shifted closer to the leading and trailing edge. The displacements at the leading and trailing edges were 0.352 and 0.1500, respectively. For cases I through V, this topology had the greatest trailing edge deflection. The reason that the other cases had smaller trailing edge deflections was because the actuator load was located far from the trailing edge. Moving the actuator load closer to where the desired deflection occurred led to a bigger deflection.

Case VI: Thick Skin with Supports at Bottom Corners of Wing Box with "Pressure" Force

Figure 20 illustrates the final topology generated with the normal HCA code.

Figure 20: Airfoil configuration of thick skin, mass fraction = 0.318, supports at bottom corners of wing box (green points), and actuator loads at red point points. There was a unit force per unit area vertically loading.

This topology does not represent a viable solution. The pressure force, although the same in magnitude for each force, was of much greater quantity than the actuator force. As a result, the pressure force overpowered the actuator force. Material was placed in the structure primarily as a result of the pressure force. Due to the large amount of pressure forces, the deflections at the leading and trailing edge were 55.6 and 19.58, respectively. These deflections are too large to be considered. A deflection of 55.6 units is larger than the design domain. A better model must be created for this initial design. An initial compliance minimization possibly could be run in order to create a geometry sufficient for supporting the aerodynamic loads. Once this geometry was known, it could be used as an initial geometry in a compliant mechanism analysis. The structure from the

compliant minimization study could be forced to always be present. The compliant mechanism analysis would create more structure which when loaded with actuator force would deflect the leading and trailing edges.

Summary

Table 1 summarizes the different initial designs for the normal HCA algorithm investigation. By examining the figures of the topologies generated and table 1, case III with the increased mass fraction appears to be the best initial design domain. A better design domain may result from combining case III with case V, or an increased mass ratio with actuator loads closer to the desired deflection.

Case	Skin Thickness	Mass Fraction	Support Location	Actuator Load Location	Leading Edge Deflection	Trailing Edge Deflection
I	Thick	0.318	Bottom	Normal	0.268	-0.0356
II	Thin	0.229	Bottom	Normal	Due to bad results, not considering these deflections	
III	Thick	0.352	Bottom	Normal	0.368	0.1187
IV	Thick	0.318	Top	Normal	0.268	-0.0356
V	Thick	0.318	Bottom	Extended	0.352	0.1500
VI	Thick	0.3180	Bottom	Normal	55.6	19.58

Table 1: Summary of values obtained from normal HCA algorithm. Thick skin corresponds to skin 3 elements thick. Thin skin corresponds to skin 1 element thick. Bottom actuator load location corresponds to fixed nodes as indicated in figure 8. Top actuator load location corresponds to fixed nodes located at the top 2 corners of the wing box. Normal actuator load location corresponds to the actuator loads as shown in figure 8. Extended actuator load location corresponds to the actuator loads being 10 units closer to the leading and trailing edge.

4.2 – Support Optimization with Compliance Minimization

The support optimization simulations for structures undergoing compliance minimization verified that the support adaptation to the HCA code was functioning correctly for structures. The first structure studied with a load at the top center and the support region along the bottom had supports placed below the applied load as seen in figure 21. This makes sense since the areas of highest stress and strain are directly below the applied force. To reduce the compliance and to increase the rigidity of the design, supports and material were placed directly below the applied load.

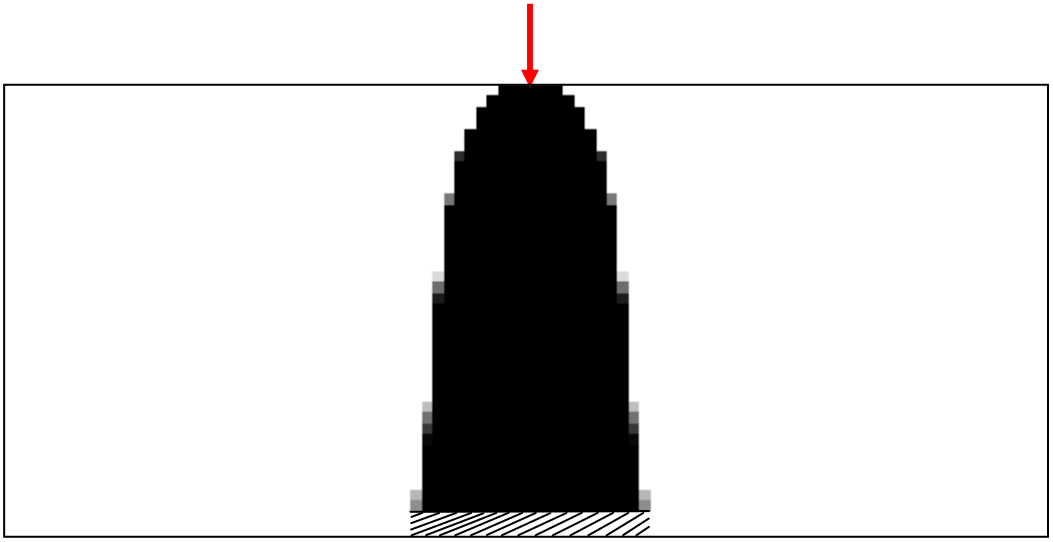

Figure 21: Result of support optimization on rectangular design domain loaded from the top.

The next structure studied was modeled so that supports could not be placed below the applied load. Therefore, more compliance in the structure would result because of the inherent more flexible constraints placed on the design. Nevertheless, again the algorithm placed material and supports in such a way to minimize the compliance as much as possible. Since the load was placed in the center of the design domain both horizontally and vertically a symmetrical structure in both planes is expected. Figure 22 illustrates that such a design was derived by the algorithm. Since supports could not be placed directly in line with the applied load, as was the case in the

previous example and would be the design leading to the stiffest structure, they were placed in the next preferable spot. Supports were placed as close to above and below the applied load as possible. Since the supports were placed on the top and bottom of the optimization region, the component of the applied load felt by the supports is as large as possible. This results in a stiffer structure. If the supports were placed closer to the center that separates the top of the design domain from the bottom of the design domain, higher stresses and strains as well as deflections would result due to the increased mechanical advantage that would result from that design.

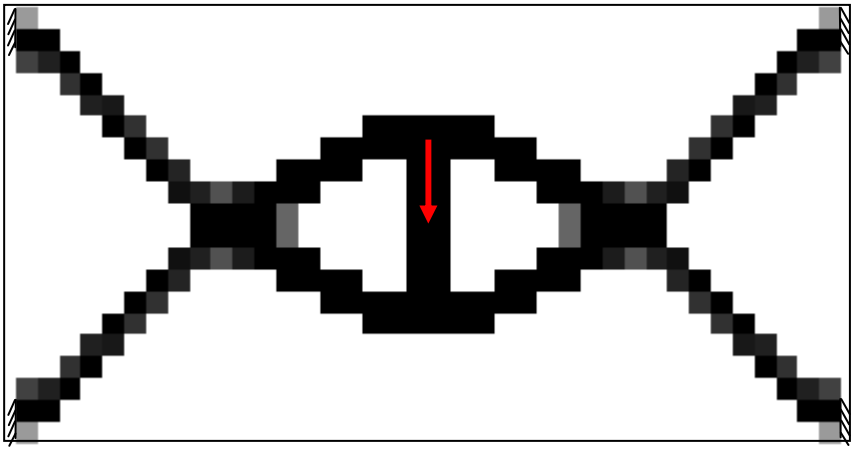

Figure 22: Result of support optimization on rectangular design domain loaded at center with side support optimization regions.

The final compliant minimization study conducted was a rectangular design domain loaded at the center of the top with only one side of region allowed to form supports. The result was again a structure with supports at the extremes of the support regions, thus leading to less deflection of the structure as can be seen in figure 23. It is interesting to notice that the top cell and bottom 2 cells in the support optimization region were under such high stress and were undergoing enough strain to warrant not only support placement but also material placement. As was the case most often in this study, cells

where supports were placed were often stiff enough that the algorithm did not place mass in those cells. The structure was stiffer if the mass that would be placed in those cells was placed in other locations. However, it this specific example as could be possible in other examples, support and material was necessary to minimize the compliance.

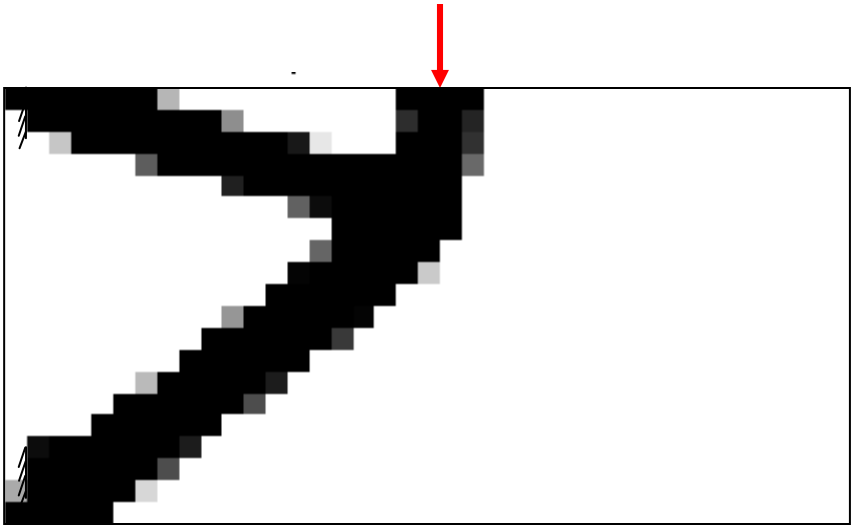

Figure 23: Result of support optimization on rectangular design domain loaded at the center of the top with a support optimization region on one side only.

4.3 – Support Optimization with simple Compliant Mechanisms

Support optimization with the force inverter also led to better results from the HCA code. The output displacement of the inverter using the conventional initial design was -0.3525. The output displacement of the inverter using the new code that makes use of support optimization was -0.3818. The new method led to an 8.31 % increase in the output displacement. The results shown in figure 24 illustrate that simply placing supports around the edges of a design domain will not always result in the largest displacements possible.

Figure 24: Result of support optimization on force inverter. The figure at the top is the topology generated using conventional support locations along the edge of the design domain. The figure at the bottom is the topology generated using the support optimization algorithm.

Moving the supports within the design domain can lead to larger displacements at the desired locations. Also note that no convergence was achieved for either the conventional or the new support optimization method. The topologies generated above were the result of 100 iterations.

4.4 – Support Optimization with NACA 0012 Airfoil

The topology generated via the support optimization of a NACA 0012 airfoil was a result of an initial design exactly the same as case I of the normal HCA algorithm studies but with regions around a wing box area for support optimization rather than prescribed supports. For this reason, the topology generated as shown in figure 25 will be compared to case I of the normal HCA algorithm study.

Figure 25: Airfoil configuration of thick skin, mass fraction = 0.239, support optimized (green points), and actuator loads at red point points.

This topology looks similar to the topology generated in case I (figure 15). One major difference is that the optimized support locations solved for by the algorithm differ from the fixed points that were arbitrarily chosen. While these support locations differ, the original guess of the lower corners of the wing box as attachment points was confirmed to be a reasonable estimate.

Figure 26 below simulates two central wing box regions and roughly approximates where on the wing box regions attachment points were located for the original HCA model and the support optimization model. The wing box on the left is the wing box from figure 15 with the red squares indicating approximately where fixed nodes were set. The wing box on the right indicates the approximate regions where the algorithm solved for supports. The deflections on the leading and trailing edge for the airfoil in figure 25 were 0.408 and 0.0654, respectively. This corresponds to a 52.2 % increase in deflection at the leading edge and an 83.7 % increase in deflection at the trailing edge.

Figure 26: Two representations of the support location on the wing box for the normal HCA code (left) and the support optimization HCA code (right). Note that only the outside region is eligible for support placement.

There are two main reasons why the deflections are much greater at the leading and trailing edges with the support optimization. First of all, the support location was optimized to make the stiffest support structure possible. As a result, the actuator loads had stiffer locations to push off of, and therefore greater output displacements occured. Secondly, the support optimization scheme was programmed to solve for 5 % of the possible region as support cells. This resulted in more attachment points than were programmed for case I of the normal HCA code. The increase in number of attachment points also led to a stiffer structure to push off of and thus greater output displacements. Overall this model was successful, achieving greater displacements in the downward direction at the leading and trailing edges of the NACA 0012 airfoil

5.0 – Discussion and Conclusions

Using the original HCA algorithm, a study was conducted to examine how varying several initial conditions on a NACA 0012 initial geometry design domain would affect the output displacements at the leading and trailing edge. From this study it was seen that in general, using a slightly larger mass fraction of approximately 0.35 and moving actuators closer to the leading and trailing edge led to better displacements.

Before examining the NACA 0012 airfoil with the support optimization code, several studies were conducted to verify that the new code worked correctly. Studies in both compliance minimization for stiff structure synthesis as well as simple compliant mechanism synthesis illustrated that the code worked and added an improvement onto the normal HCA algorithm.

As expected based on the validated results in both compliance minimization and simple compliant mechanisms, the support optimization improved the output displacements for both the leading and trailing edges for the NACA 0012 airfoil. Support locations solved for by the algorithm were in the vicinity of the supports arbitrarily chosen for the initial HCA study without support optimization. However, a slight increase in the number of supports as well as placing them at the optimal locations led to a noticeable improvement in output location. The goal of this study was to create a stiffer support structure by minimizing compliance in a support region in order to give a firmer base with which actuator loads could push off. This led to an increase in displacement at the desired output locations from the actuation loads as planned.

Despite these successes, this method is still far from useful in designing elastic wings for aircraft. The first and most easily correctable characteristic of this method

which currently hurts its usefulness is the use of normalized values in the study. No system of units was assigned for this study. Forces, stiffness, and material properties were all given an order of magnitude of 1. This problem can easily be rectified, however, if the problem remains linear. A simple scaling of variables will not result in any changes in the topologies generated.

A second issue is the lack of flow modeling in this study. At no point was flow over the airfoil modeled. This flow can cause large and sometimes localized stresses over the skin of the airfoil. Even simple attempts to model a static atmosphere failed because the loading was much greater than the actuation load. A computational fluid dynamics analysis could be added to the iterative loop of the HCA method before the finite element analysis in order to analyze the stresses on the airfoil due to the flow and to pass them into the finite element analysis. An initial compliance minimization analysis to create an initial geometry suitable to handle the loading from the flow, coupled with larger actuation loads, could be used to overcome the pressure forces and create a valid topology.

A third obstacle is that only a linear finite element analysis was used in this study. Since large deformations may occur in a compliant mechanism, it is generally not a safe assumption that a linear finite element analysis will be sufficient. To make this model more accurate for real world applications, a non-linear finite element analysis should be used in lieu of the linear analysis. This, combined with actual values for forces and other properties in the model as opposed to normalized values, will enable this method to handle non-linear situations.

Finally, a major complication that would have to be overcome before this method could be used to design the structure for a three dimensional wing is the computational time. For the new method that combines HCA with support optimization, the computational time for a 40 by 240 design domain is approximately one-half hour per iteration. Only approximately 50 iterations were used for the NACA 0012 support optimization due to the large amount of time necessary for the algorithm to run. In order to use this method for an elastic wing, a three dimensional analysis would be necessary. This would greatly increase the computing time. In addition, more iterations would be needed to find a solution that converged or that was closer to converging on a final solution. A topology with as much intermediate density material remaining in it as was the case in this study would not be acceptable. Also, a finer mesh size, and thus more elements, also would most likely be necessary to get better topologies, which would significantly increase the computation time. All of these increases in computation time combined with the increase in computation time due to the computational fluid dynamics modeling would lead to such a large computation time that this method would be impractical for even supercomputers.

In order to improve the computation time the computer code could be modified with efficiency in mind. Also, faster computing languages and formats, such as C, could be used rather than MATLAB to try and decrease computing time.

Despite these setbacks, this study did lead to an improvement in the HCA algorithm. This method will be viable for designing elastic wings in the future after the obstacles mentioned above have been resolved.

6.0 – References

[1] Maute, K., "An Aeroelastic Topology Optimization Approach for Adaptive Wing Design," *45th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, &Materials Conference*, Paper 2004-1805, 2004, pp. 1-2.

[2] Weisshaar, Terry A., "Aeroelastic tailoring for energy efficient morphging aircraft – finding the right stuff," *ICASE NASA/LsRC,* 2001, pp. 1-6.

[3] Weisshaar, Terry A., Crossley, William A., Roth, Brian, and Peters, Christopher, "Use of design methods to generate and develop missions for morphing aircraft," *American Institude of Aeronautics and Astronautics*, Paper 2002-5468, 2002, pp. 1-11.

[4] Tovar, A., Patel, N., A.K. Kaushik, G. L., and Renaud, J., "Hybrid Cellular Automata: a biologically inspired structural optimization technique," *Proceedings of the 10th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization,* No. AIAA 2004-4558, Albany, New York, Aug 30-Sep. 1 2004.

[5] Bendsoe, M.P. "Optimal shape design as a material distribution problem," *Struct Optim.*, Vol. 1, 1989, pp. 193-200.

[6] Zhou, M. and Rozvany, G.I.N., "The COC algorithm, part II: Topological, geometry and generalization shape optimization," *Comput. Methods Appl. Mech. Engrg.,* Vol 89, 1991, pp. 197-224.

[7] Patel, Neil M. and Renaud, John E., "Compliant Mechanism Design using the Hybrid Cellular Automaton Method," *46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, &Materials Conference*, Paper 2005-2276, 2005, pp. 1-12.

[8] Bendose, M.P. and Sigmund, O., *Topology Optimization: Theory, Methods, and Applications*, Springer, 2004, pp. 94-108.

[9] Patel, Neil M., Tovar, A., Renaud, and John E., " Compliant Mechanism Design using the Hybrid Cellular Automaton Method," presentation for *1st AIAA Multidisciplinary Design Optimization Specialist Conference*, 2005, p. 4.

[10] Buhl, T, "Simultaneous topology optimization of structure and supports," *Structural Multdisc Optim*, Vol 23, 2002, pp. 336-346.

[11] Tovar, A., *Bone Remodeling as a Hybrid Cellular Automoaton Optimization Process,* Ph. D. thesis, University of Notre Dame, 2004.

7.0 – Appendix

******NOTE: A majority of this code was written previous to this study. The entire code has been included although most of it was not the work of this author. Portions of the code which the author wrote have been indicated via comments. In the main program the portion written by the author is in one large block and is indicated at the beginning and the end by comments (Pages 48-54). In the support location code the alterations are mixed throughout and thus lines of code written by the author are indicated by a '%*' on the right. Appendix C is not the work of the author at all but rather was code provided by the University of Notre Dame.******

7.1 – Appendix A: Main HCA Code with Airfoil Adaptation

```
function varargout = ndopti(varargin)
% NDOPTI M-file for ndopti.fig
%      NDOPTI, by itself, creates a new NDOPTI or raises the existing
%      singleton*.
%
%      H = NDOPTI returns the handle to a new NDOPTI or the handle to
%      the existing singleton*.
%
%      NDOPTI('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in NDOPTI.M with the given input arguments.
%
%      NDOPTI('Property','Value',...) creates a new NDOPTI or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before ndopti_OpeningFunction gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to ndopti_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help ndopti

% Last Modified by GUIDE v2.5 22-Sep-2005 01:33:06

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @ndopti_OpeningFcn, ...
                   'gui_OutputFcn',  @ndopti_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
```

```
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before ndopti is made visible.
function ndopti_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to ndopti (see VARARGIN)

% Choose default command line output for ndopti
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes ndopti wait for user response (see UIRESUME)
% uiwait(handles.ndoptifig);


% --- Outputs from this function are returned to the command line.
function varargout = ndopti_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes during object creation, after setting all properties.
function ndoptifig_CreateFcn(hObject, eventdata, handles)
clear 'all'
disp('------------------------------------------------------')
disp('welcome to ndopti ver 7.0 updated 4-03-06')
cl = clock;
hr = [num2str(cl(4)),'h ',num2str(cl(5)),'min ',num2str(cl(6)),'sec'];
str = [date,'   ',hr];
disp(str)
global x fixedU fixedF Ymodul Pratio pPower nelx nely springK q             %*i
added springK,q
Ymodul = 1.0;
Pratio = 0.3;
pPower = 3.0;

function nelx_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
```

```
function nelx_Callback(hObject, eventdata, handles)

function nely_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function nely_Callback(hObject, eventdata, handles)

function listbox1_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function domainregion_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function domainshape_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function domainshape_Callback(hObject, eventdata, handles)

function x1shape_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function x1shape_Callback(hObject, eventdata, handles)

function x2shape_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function x2shape_Callback(hObject, eventdata, handles)

function y1shape_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
```

```
function y1shape_Callback(hObject, eventdata, handles)

function y2shape_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function y2shape_Callback(hObject, eventdata, handles)

% --- Executes on button press in domain0button.
function domain0button_Callback(hObject, eventdata, handles)
global targets x nelx nely springK q                                    %*
nelx = str2num(get(handles.nelx,'String'));
nely = str2num(get(handles.nely,'String'));
dens = str2num(get(handles.dens,'String'));
sens = str2num(get(handles.sens,'String'));
if dens < 0
    x = rand(nely,nelx)*(-dens);
    x(x>0.999)=0.999;
    x(x<0.001)=0.001;
    mycolor = [0 1 1]*mod(1+dens,1);
    %mycolor = [0 1 1]*(1+dens);
    %mycolor(mycolor<0) = 0;
else
    x = ones(nely,nelx)*dens;
    mycolor = [1 1 1]*(1-dens);
    mycolor(mycolor<0) = 0;
end



q = zeros(size(x));                                                     %*
for i = 1:nelx
    q(1:4,i) = .05;
    q(37:40,i) = .05;
end
%q(19:20,1) = .999;  %half inverter
% for i = 1:nelx
%     q(nely,i) = 0.2;
% end

x(x>1) = 1;
targets = ones(nely,nelx)*sens;
hold on; grid on;
axis([-1 nelx+1 -1 nely+1]);
axis equal; axis ij;
hrec = rectangle('Position', [0, 0, nelx, nely],'Facecolor',mycolor);
hrecf = get(handles.resetdomain,'UserData');
set(handles.resetdomain,'UserData',[hrecf; hrec])


function domaincoord_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
```

```
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function domaincoord_Callback(hObject, eventdata, handles)

% --- Executes on button press in domainfbutton.
function domainfbutton_Callback(hObject, eventdata, handles)
global x targets nelx nely fixedF fixedU springK q                              %*
vshape = get(handles.domainshape,'Value');
dens = str2num(get(handles.dens,'String'));
sens = max(0, str2num(get(handles.sens,'String')));
if dens < 0
    mycolor = [0 1 1]*mod(1+dens,1);
    %mycolor = [0 1 1]*(1+dens);
    %mycolor(mycolor<0) = 0;
else
    mycolor = [1 1 1]*(1-dens);
    mycolor(mycolor<0) = 0;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%ALTERATIONS MADE BY DAVID LETTIERI BEGIN HERE%%%%%%%%%%%%%%%%

%David Lettieri altered below, 4/26/06
%for column mistake, search 'sort column mistake'
if (vshape == 1) || (vshape == 2)
    x1shape = eval(get(handles.x1shape,'String'));
    x2shape = eval(get(handles.x2shape,'String'));;
    y1shape = eval(get(handles.y1shape,'String'));;
    y2shape = eval(get(handles.y2shape,'String'));;
    xpos = min(x1shape, x2shape);
    ypos = min(y1shape, y2shape);
    sidex = abs(x1shape - x2shape);
    sidey = abs(y1shape - y2shape);
    hrec = rectangle;
    set(hrec,'Position',[xpos, ypos, sidex, sidey],'FaceColor',mycolor);
    newx = x;
    newtargets = targets;
    if dens < 0
        newx(ypos+1 : ypos+sidey, xpos+1 : xpos+sidex) = rand(sidey,
sidex)*abs(dens);
        newx(newx>0.999)=0.999;
        newx(newx<0.001)=0.001;
    else
        newx(ypos+1 : ypos+sidey, xpos+1 : xpos+sidex) = dens;
    end
    newtargets(ypos+1 : ypos+sidey, xpos+1 : xpos+sidex) = sens;
    if vshape == 1 %rectangle
        set(hrec,'Curvature',0);
    elseif vshape == 2 %circle
        set(hrec,'Curvature',1);
        r = min(sidex,sidey)/2;
```

```
        for elx=0:round(r)
            for ely=0:round(r)
                R = sqrt((r-elx)^2 + (r-ely)^2);
                if (R-r)>0.5
                    newx(ypos+ely+1,xpos+elx+1) = x(ypos+ely+1,xpos+elx+1);
                    newx(ypos+ely+1,xpos+sidex-elx) = x(ypos+ely+1,xpos+sidex-
elx);
                    newx(ypos+sidey-ely,xpos+elx+1) = x(ypos+sidey-
ely,xpos+elx+1);
                    newx(ypos+sidey-ely,xpos+sidex-elx) = x(ypos+sidey-
ely,xpos+sidex-elx);
                    newtargets(ypos+ely+1,xpos+elx+1) =
targets(ypos+ely+1,xpos+elx+1);
                    newtargets(ypos+ely+1,xpos+sidex-elx) =
targets(ypos+ely+1,xpos+sidex-elx);
                    newtargets(ypos+sidey-ely,xpos+elx+1) = targets(ypos+sidey-
ely,xpos+elx+1);
                    newtargets(ypos+sidey-ely,xpos+sidex-elx) =
targets(ypos+sidey-ely,xpos+sidex-elx);
                end
            end
        end
    end

    x = newx;
    x(x>1) = 1;
    targets = newtargets;
    hrecf = get(handles.resetdomain,'UserData');
    set(handles.resetdomain,'UserData',[hrecf;hrec]);
elseif vshape == 3
    x1shape = eval(get(handles.x1shape,'String')); % Lee las coordenadas
    x2shape = eval(get(handles.x2shape,'String'));;
    y1shape = eval(get(handles.y1shape,'String'));;
    y2shape = eval(get(handles.y2shape,'String'));;
    xpos = min(x1shape, x2shape); % Calcula la posicion del rectangulo
    ypos = min(y1shape, y2shape);
    chord = abs(x1shape - x2shape);
    xairfoil = 0:chord-1;       %is this -1 right
    t = .12*max(xairfoil);
    yu = 5*(0.29690.*(xairfoil/chord).^0.5 - 0.12600.*(xairfoil/chord)-
0.35160.*(xairfoil/chord).^2 + ...
        0.2843.*(xairfoil/chord).^3 - 0.10150.*(xairfoil/chord).^4);
    yu = round(yu.*t);              %take round out
    yl = -yu;
    len = length(xairfoil);
    yuyl = yu;
    %fall 06 adjustments to fix patch plot - doesn't correct x for patch -
    %too complicated at moment (need to go 1 more element at right)
    xairfoilpatch = xairfoil;
    yuylpatch = yuyl+1;
    for counter = 1:len
        xairfoil(len+counter) = xairfoil(len+1-counter);
        yuyl(len+counter) = yl(len+1-counter)-1;
        yuylpatch(len+counter) = yl(len+1-counter)-1;
    end
```

```
    xairfoil = xairfoil+xpos;           %translates the aifoil to xpos
    yuyl = round(yuyl) + ypos;%don't need to ADD ONE        %makes the airfoil
shape lie on integer
                                        %nodes and also translates shape to
                                        %ypos - the plus one on yuyl is
                                        %necessary to make sure that the
                                        %airfoil is centered - look into
                                        %this - 2/11/07 - checked,
                                        %everything looks good


%%2/11/07patch does not account for the three thick celled skin, other than
that,
%%it works

%%%%%%%%%%%%%%%%%code to check patch%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%save variables in appropriate places in code
    %save xairfoilpatch xairfoilpatch
    %save yuylpatch yuylpatch
    %save x x

    %%then after run program, type into matlab
    %open  xairfoilpatch xairfoilpatch
    %open yuylpatch yuylpatch
    %open x x
    %plot(xairfoilpatch,yuylpatch,'.')
    %hold on
    %axis equal
    %mycolor = [.1 .2 .69];
    %hrec = patch(xairfoilpatch,yuylpatch,mycolor)
    %figure(2)
    %himgc = imagesc(-x,[-1 0]);
    %colormap(gray); axis equal; axis tight; axis off;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


    xairfoilpatch = xairfoil;
    yuylpatch = round(yuylpatch)+ypos;

    hrec = patch(xairfoilpatch,yuylpatch,mycolor);
    newx = x; % Redefine las variables para los elementos
    newtargets = targets;

    %making outside region passive, inside region passive- i believe i
    %added active1 line codes down far below shown here:
        %active1 = (x==1);
        %feax(find(active1)) = 1.0;

    %NEED TO EDIT NEGATIVE DENSITIES - what do negative densities do?
    if dens < 0 % Si la densidad es negativa, el rectangulo tiene densidad
aleatoria
        newx(ypos+1 : ypos+sidey, xpos+1 : xpos+sidex) = rand(sidey,
sidex)*abs(dens);
        newx(newx>0.999)=0.999;
        newx(newx<0.001)=0.001;
```

```
    else % Si la densidad es positiva, el rectangulo tienen ese valor de
densidad
        for i = 0:nelx-1
            for j = 0:nely-1
                %inside of inner region is variable
                if (i)>xpos && (i)<(xpos+chord) && (j)< yuyl(i+1-xpos) && (j) >
yuyl(2*len-(i-xpos))
                    newx(j+1,i+1) = dens;
                    newtargets(j+1,i+1) = sens;
                %outside of inner region is always passive
                else
                    newx(j+1,i+1) = 0;
                    newtargets(j+1,i+1) = 0;   %this line may be wrong
                end
            end
        end
    end
    %boundary that defines inner and outer region is always active - also
    %putting in a static force of 1 in x and y direction at each upper
    %left node of skin

    %fixed loads
    loadcase = 1;   %simultaneous loads
    % color of the arrow
    tc = mod(dec2bin(loadcase+8),8);
    tc = fliplr(tc);
    harrowcolor = [tc(1) tc(3) tc(2)];
   %fixed displacements
    r = (1 - 1/loadcase)/2;
    g = (1 - 1/loadcase)/2;
    b = (1 - 1/loadcase)/2;
    mycolor = [r g 0];

    %defining skin of airfoil as always active - thick skin (modified fall
    %06)
    for i = 1:2*len
        newx(yuyl(i):yuyl(i)+2,xairfoil(i)+1) = 1.0;      %make this equal to 1
- then will be active always
        newtargets(yuyl(i):yuyl(i)+2,xairfoil(i)+1) = sens;
    end
    newx(ypos-2:ypos+3,2) = 1.0;    %ADDED THIS LINE TO FILL IN LEADING EDGE GAP

    %%%%%%%%%%%%%%%%%%force on skin due to atmosphere%%%%%%%%%%%%%%%%%%%%%%
    %note, the plus one on the yuyl(i) is so that the force is on the
    %outside of the finite element (necessary since the first for loop
    %(1:len) is the bottom of the airfoil

    for i = 1:len
        %fixed loads
        fixedF = [fixedF; 2*((nely+1)*xairfoil(i)+(yuyl(i)+3)),-1, loadcase];
%ydirection force

plot_arrow(xairfoil(i),yuyl(i)+3,xairfoil(i),yuyl(i)+1,'color',harrowcolor,'face
color',harrowcolor,'edgecolor',harrowcolor,'linewidth',2,'headwidth',0.25,'headh
eight',0.33);
    end
```

```
    for i = len+1:2*len
        %fixed loads
        fixedF = [fixedF; 2*((nely+1)*xairfoil(i)+(yuyl(i))),1, loadcase];
%ydirection force

plot_arrow(xairfoil(i),yuyl(i),xairfoil(i),yuyl(i)+2,'color',harrowcolor,'faceco
lor',harrowcolor,'edgecolor',harrowcolor,'linewidth',2,'headwidth',0.25,'headhei
ght',0.33);
    end


%          fixedF = [fixedF; [22 70 114 158]',-1+zeros(4,1),
loadcase+zeros(4,1)]; %ydirection force
%          fixedF = [fixedF; [5036 5078 5120 5162]',1+zeros(4,1),
loadcase+zeros(4,1)]; %ydirection force
%          springK = [springK; [22 70 114 158]',.1+zeros(4,1),
loadcase+zeros(4,1)]; %ydirection force
%          springK = [springK; [5036 5078 5120 5162]',.1+zeros(4,1),
loadcase+zeros(4,1)]; %ydirection force


%           %for 240X40, four displacements on trailing edge, 7 on leading
%           %edge (4 on top, 4 on bottom)
%           fixedF = [fixedF; [42 132 216 300 116 196 276]',-9+zeros(7,1),
loadcase+zeros(7,1)]; %ydirection force
%           fixedF = [fixedF; [5036 5078 5120 5162]',18+zeros(4,1),
loadcase+zeros(4,1)]; %ydirection force
%           springK = [springK; [42 132 216 300 116 196 276]',.1+zeros(7,1),
loadcase+zeros(7,1)]; %ydirection force
%           springK = [springK; [5036 5078 5120 5162]',.1+zeros(4,1),
loadcase+zeros(4,1)]; %ydirection force


%%%%%%%%%no torque box for attachment optimization
%     %torque box
%     mycolortb = [1 1 1]*(1-.999);
%     hrec = rectangle;
%     set(hrec,'Position',[chord/4, ypos-.2*nely, chord/4 ,
0.4*nely],'FaceColor',mycolortb);
%     newx(ypos-.2*nely+1 : ypos-.2*nely+.4*nely, chord/4+1 : chord/2) = 1.0;
%
     %attachment optimization                                              %*
    q = zeros(size(x));                                                   %*
    q(ypos-.2*nely+1 : ypos-.2*nely+.4*nely, chord/4+1 : chord/2) = 0.05; %*
    q(ypos-.2*nely+2 : ypos-.2*nely+.4*nely-1, chord/4+2 : chord/2-1) = 0; %*

    for i = 1:len/4
        %fixed loads
        fixedF = [fixedF; 2*((nely+1)*xairfoil(i)+(yuyl(i)+3)),1, 2];
%ydirection force

plot_arrow(xairfoil(i),yuyl(i)+3,xairfoil(i),yuyl(i)+1,'color',harrowcolor,'face
color',harrowcolor,'edgecolor',harrowcolor,'linewidth',2,'headwidth',0.25,'headh
eight',0.33);
```

```
        %fixed spring constants
        springK = [springK; 2*((nely+1)*xairfoil(i)+(yuyl(i)+3)), 0.001, 2];
        plot_spring(xairfoil(i), yuyl(i)+3, 3, mycolor); %erased htri, is that
ok?
    end

    for i = 3*len/4:len
        %fixed loads
        fixedF = [fixedF; 2*((nely+1)*xairfoil(i)+(yuyl(i)+3)),1, 2];
%ydirection force

plot_arrow(xairfoil(i),yuyl(i)+3,xairfoil(i),yuyl(i)+1,'color',harrowcolor,'face
color',harrowcolor,'edgecolor',harrowcolor,'linewidth',2,'headwidth',0.25,'headh
eight',0.33);
        %fixed spring constants
        springK = [springK; 2*((nely+1)*xairfoil(i)+(yuyl(i)+3)), 0.001, 2];
        plot_spring(xairfoil(i), yuyl(i)+3, 3, mycolor); %erased htri, is that
ok?
    end

    for i = len+1:5/4*len+1
        %fixed loads
        fixedF = [fixedF; 2*((nely+1)*xairfoil(i)+(yuyl(i))),1, 2]; %ydirection
force

plot_arrow(xairfoil(i),yuyl(i),xairfoil(i),yuyl(i)+2,'color',harrowcolor,'faceco
lor',harrowcolor,'edgecolor',harrowcolor,'linewidth',2,'headwidth',0.25,'headhei
ght',0.33);
        %fixed spring constants
        springK = [springK; 2*((nely+1)*xairfoil(i)+(yuyl(i))), 0.001, 2];
        plot_spring(xairfoil(i), yuyl(i), 3, mycolor); %erased htri, is that ok?
    end
    for i = 7/4*len+1:2*len
        %fixed loads
        fixedF = [fixedF; 2*((nely+1)*xairfoil(i)+(yuyl(i))),1, 2]; %ydirection
force

plot_arrow(xairfoil(i),yuyl(i),xairfoil(i),yuyl(i)+2,'color',harrowcolor,'faceco
lor',harrowcolor,'edgecolor',harrowcolor,'linewidth',2,'headwidth',0.25,'headhei
ght',0.33);
        %fixed spring constants
        springK = [springK; 2*((nely+1)*xairfoil(i)+(yuyl(i))), 0.001, 2];
        plot_spring(xairfoil(i), yuyl(i), 3, mycolor); %erased htri, is that ok?
    end

    x = newx; % Quedan todos los valores de densidad en cells
    x(x>1) = 1;
    targets = newtargets;
    hrecf = get(handles.resetdomain,'UserData');
    set(handles.resetdomain,'UserData',[hrecf;hrec]);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%END OF ALTERATIONS TO MAIN CODE BY DAVID LETTIERI%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function fixeddoftype_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function fixeddoftype_Callback(hObject, eventdata, handles)


function x1fixed_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function x1fixed_Callback(hObject, eventdata, handles)

function y1fixed_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function y1fixed_Callback(hObject, eventdata, handles)

function x2fixed_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function x2fixed_Callback(hObject, eventdata, handles)

function y2fixed_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function y2fixed_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function fixedregion_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
```

```
end

function fixedregion_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function loadsregion_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function loadsregion_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function x1loads_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function x1loads_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function y1loads_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function y1loads_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function x2loads_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function x2loads_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function y2loads_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function y2loads_Callback(hObject, eventdata, handles)
```

```
% --- Executes during object creation, after setting all properties.
function fxedt_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function fxedt_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function fyedt_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function fyedt_Callback(hObject, eventdata, handles)

% --- Executes on button press in pickloads.
function pickloads_Callback(hObject, eventdata, handles)
vloadsregion = get(handles.loadsregion,'Value');
if vloadsregion==1
    coord = ginput(1);
    set(handles.x1loads,'String',round(coord(1,1)));
    set(handles.y1loads,'String',round(coord(1,2)));
elseif vloadsregion==2
    coord = ginput(1);
    set(handles.x1loads,'String',round(coord(1,1)));
    set(handles.y1loads,'String',round(coord(1,2)));
    coord = ginput(1);
    set(handles.x2loads,'String',round(coord(1,1)));
    set(handles.y2loads,'String',round(coord(1,2)));
end


% --- Executes on button press in applyloads
function applyloads_Callback(hObject, eventdata, handles)
global x fixedF
vloadsregion = get(handles.loadsregion,'Value');
fx = eval(get(handles.fxedt,'String'));
fy = eval(get(handles.fyedt,'String'));
loadcase = eval(get(handles.loadcase,'String'));
nelx = size(x,2);
nely = size(x,1);
harrow = get(handles.resetloads,'UserData');
x1loads = eval(get(handles.x1loads,'String'));
y1loads = eval(get(handles.y1loads,'String'));
% color of the arrow
tc = mod(dec2bin(loadcase+8),8);
tc = fliplr(tc);
harrowcolor = [tc(1) tc(3) tc(2)];
if vloadsregion == 1 %individual node
    harrow = [harrow; plot_arrow(x1loads, y1loads, x1loads+sign(fx)*2,
y1loads+sign(fy)*2,'color',harrowcolor,'facecolor',harrowcolor,'edgecolor',harro
wcolor,'linewidth',2,'headwidth',0.25,'headheight',0.33)];
```

```
    if fx
        fixedF = [fixedF; 2*(y1loads+1) + 2*x1loads*(nely+1) - 1, fx, loadcase];
    end
    if fy
        fixedF = [fixedF; 2*(y1loads+1) + 2*x1loads*(nely+1)    , fy, loadcase];
    end
elseif vloadsregion == 2 %multiple nodes
    x2loads = eval(get(handles.x2loads,'String'));
    y2loads = eval(get(handles.y2loads,'String'));
    xmaxrec = max(x1loads,x2loads);
    xminrec = min(x1loads,x2loads);
    ymaxrec = max(y1loads,y2loads);
    yminrec = min(y1loads,y2loads);
    for nodex = xminrec:xmaxrec
        for nodey = yminrec:ymaxrec
            harrow = [harrow; plot_arrow(nodex, nodey, nodex+sign(fx)*2,
nodey+sign(fy)*2,'color',harrowcolor,'facecolor',harrowcolor,'edgecolor',harrowc
olor,'linewidth',2,'headwidth',0.25,'headheight',0.33)];
            if fx
                fixedF = [fixedF; 2*(nodey+1) + 2*nodex*(nely+1) - 1, fx,
loadcase];
            end
            if fy
                fixedF = [fixedF; 2*(nodey+1) + 2*nodex*(nely+1)    , fy,
loadcase];
            end
        end
    end
end
set(handles.resetloads,'UserData',harrow);

% --- Executes on button press in resetloads.
function resetloads_Callback(hObject, eventdata, handles)
global fixedF
harrow = get(handles.resetloads,'UserData');
delete(harrow);
set(handles.resetloads,'UserData',[]);
fixedF = [];


% --- Executes on button press in checkden
function checkden_Callback(hObject, eventdata, handles)
global x
disp('checking x...');
figure;
himgc = imagesc(-x,[-1 0]);
colormap(gray); axis equal; axis tight; axis off;


% --- Executes on button press in checksen.
function checksen_Callback(hObject, eventdata, handles)
global targets
disp('checkin sensibilities...');
figure;
hcolor = max(1,max(max(targets)));
imagesc(-targets,[-hcolor 0]);
```

```
colormap(gray); axis equal; axis tight; axis off;


% --- Executes on button press in checksupports
function checksupports_Callback(hObject, eventdata, handles)
global fixedU
disp('checking supports...')
fixedU

% --- Executes on button press in checkloads.
function checkloads_Callback(hObject, eventdata, handles)
global fixedF
disp('checking loads...')
fixedF

% --- Executes during object creation, after setting all properties.
function loadcase_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function loadcase_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function ymod_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

% --- Executes on button press in ymod.
function ymod_Callback(hObject, eventdata, handles)
global Ymodul
disp('getting Ymod...')
Ymodul = eval(get(handles.ymod,'String'));


% --- Executes during object creation, after setting all properties.
function prat_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

% --- Executes on button press in prat.
function prat_Callback(hObject, eventdata, handles)
global Pratio
Pratio = eval(get(handles.prat,'String'));


% --- Executes on button press in actualrad.
function actualrad_Callback(hObject, eventdata, handles)
```

```
set(handles.actualrad,'Value',1);
set(handles.solidrad,'Value',0);

% --- Executes on button press in solidrad.
function solidrad_Callback(hObject, eventdata, handles)
set(handles.actualrad,'Value',0);
set(handles.solidrad,'Value',1);

% --- Executes during object creation, after setting all properties.
function penal_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

% --- Executes on button press in penal.
function penal_Callback(hObject, eventdata, handles)
global pPower
pPower = eval(get(handles.penal,'String'));

% --- Executes during object creation, after setting all properties.
function dens_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function dens_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


% --- Executes on button press in ocbutton.
function ocbutton_Callback(hObject, eventdata, handles)
top99gui


% --- Executes on button press in checkUz.
function checkUx_Callback(hObject, eventdata, handles)


% --- Executes on button press in checkUz.
function checkUy_Callback(hObject, eventdata, handles)


% --- Executes on button press in controlbutton.
function controlbutton_Callback(hObject, eventdata, handles)
ndhcagui


% --- Executes on mouse press over axes background.
function axes0_ButtonDownFcn(hObject, eventdata, handles)
```

```
% --- Executes on button press in getcoordbtn.
function getcoordbtn_Callback(hObject, eventdata, handles)
coord = ginput(1);
set(handles.x1shape,'String',round(coord(1,1)));
set(handles.y1shape,'String',round(coord(1,2)));
coord = ginput(1);
set(handles.x2shape,'String',round(coord(1,1)));
set(handles.y2shape,'String',round(coord(1,2)));


% --- Executes on button press in resetdomain.
function resetdomain_Callback(hObject, eventdata, handles)
global x targets
href = get(handles.resetdomain,'UserData');
delete(href);
set(handles.resetdomain,'UserData',[]);
x = [];
targets = [];


% --- Executes on button press in bcpick.
function bcpick_Callback(hObject, eventdata, handles)
vfixeddofregion = get(handles.fixedregion,'Value');
if vfixeddofregion==1 %individual node
    coord = ginput(1);
    set(handles.x1fixed,'String',round(coord(1,1)));
    set(handles.y1fixed,'String',round(coord(1,2)));
elseif vfixeddofregion==2 %multiple multiple
    coord = ginput(1);
    set(handles.x1fixed,'String',round(coord(1,1)));
    set(handles.y1fixed,'String',round(coord(1,2)));
    coord = ginput(1);
    set(handles.x2fixed,'String',round(coord(1,1)));
    set(handles.y2fixed,'String',round(coord(1,2)));
end

% --- Executes on button press in resetsupports.
function resetsupports_Callback(hObject, eventdata, handles)
global fixedU
htri = get(handles.resetsupports,'UserData');
delete(htri);
set(handles.resetsupports,'UserData',[]);
fixedU = [];

% --- Executes on button press in applysupports.
function applysupports_Callback(hObject, eventdata, handles)
global x fixedU
vfixedregion = get(handles.fixedregion,'Value');
vfixedx = get(handles.checkUx,'Value');
vfixedy = get(handles.checkUy,'Value');
loadcase = eval(get(handles.loadcase,'String'));
nelx = size(x,2);
nely = size(x,1);
```

```
htri = get(handles.resetsupports,'UserData');
x1fixed = eval(get(handles.x1fixed,'String'));
y1fixed = eval(get(handles.y1fixed,'String'));
ux = eval(get(handles.uxedt,'String'));
uy = eval(get(handles.uyedt,'String'));
% color of the arrow
tc = mod(dec2bin(loadcase+8),8);
tc = fliplr(tc);
tcolor = [tc(1) tc(3) tc(2)];
if ux~=0 | uy ~=0
    tcolor = tcolor/2;
end
if vfixedregion == 1 %individual node
    if vfixedx %xdir
        htri = [htri; triangle(x1fixed, y1fixed, 1, tcolor)];
        fixedU = [fixedU; 2*(y1fixed+1) + 2*x1fixed*(nely+1) - 1, ux, loadcase];
    end
    if vfixedy %ydir
        htri = [htri; triangle(x1fixed, y1fixed, 2, tcolor)];
        fixedU = [fixedU; 2*(y1fixed+1) + 2*x1fixed*(nely+1), uy, loadcase];
    end
elseif vfixedregion == 2 %multiple nodes
    x2fixed = eval(get(handles.x2fixed,'String'));
    y2fixed = eval(get(handles.y2fixed,'String'));
    xmaxrec = max(x1fixed,x2fixed);
    xminrec = min(x1fixed,x2fixed);
    ymaxrec = max(y1fixed,y2fixed);
    yminrec = min(y1fixed,y2fixed);
    for nodex = xminrec:xmaxrec
        for nodey = yminrec:ymaxrec
            if vfixedx %xdir
                htri = [htri; triangle(nodex, nodey, 1, tcolor)];
                fixedU = [fixedU; 2*(nodey+1) + 2*nodex*(nely+1) - 1, ux,
loadcase];
            end
            if vfixedy %ydir
                htri = [htri; triangle(nodex, nodey, 2, tcolor)];
                fixedU = [fixedU; 2*(nodey+1) + 2*nodex*(nely+1), uy, loadcase];
            end
        end
    end
end
set(handles.resetsupports,'UserData',htri);

% --- Executes on button press in feabutton.
function feabutton_Callback(hObject, eventdata, handles)
global x fixedU fixedF springK Ymodul Pratio pPower c q

% PARAMETERS
[nely,nelx]=size(x);
lcases = max(fixedF(:,3));

mechanism=get(handles.actcbx,'Value');

tic
if ~mechanism
```

```
    str='running fea...';
    hfig=figure; axis off
    title(str); set(hfig,'Name',str); pause(1e-6); disp(str);
    feax = x;
    if get(handles.solidrad,'Value')
        feax = ones(size(x));
    end
    passive0 = (x==0);
    active1 = (x==1);
    feax(find(passive0)) = 0.001;
    feax(find(active1)) = 1.0;
    [U]=FEA(feax, Ymodul, Pratio, pPower, fixedU, fixedF, springK);

    [KE]=localK(Ymodul,Pratio);
    c = 0.;

    for ely = 1:nely
        for elx = 1:nelx
            n1 = (nely+1)*(elx-1)+ely;
            n2 = (nely+1)* elx   +ely;
            for lc=1:lcases
                Ue = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2;
2*n1+1;2*n1+2],lc);
                C(ely,elx,lc) = x(ely,elx)^pPower*Ue'*KE*Ue;
            end
        end
    end

    c=sum(C,3)/lcases;

    imagesc(c);
    axis equal; axis tight; axis off; colorbar;
    str='Compliance';
    title(str); set(hfig,'Name',str); pause(1e-6); disp(str);
    str = [ ' Cmin: ' sprintf('%6.4f',min(min(c))) ...
        ' Cmax: ' sprintf('%6.4f',max(max(c))) ...
        ' Ctot: ' sprintf('%6.4f',sum(sum(c)))];
    title(str); pause(1e-6); disp(str);
    lcase = max(fixedF(:,3));
    if lcase>1
        for lc=1:lcase
            hfig=figure; str=['Load case ',num2str(lc)];
            imagesc(C(:,:,lc));
            axis equal; axis tight; axis off; colorbar;
            title(str); set(hfig,'Name',str); pause(1e-6); disp(str);
            str = [ ' Cmin: ' sprintf('%6.4f',min(min(C(:,:,lc)))) ...
                ' Cmax: ' sprintf('%6.4f',max(max(C(:,:,lc)))) ...
                ' Ctot: ' sprintf('%6.4f',sum(sum(C(:,:,lc))))];
                title(str); pause(1e-6); disp(str);
            end
        end

else
    str = 'running fea...';
    disp(str)
    figure; axis off
```

```
    title(str); pause(1e-6);
    feax = x;
    if get(handles.solidrad,'Value')
        feax = ones(size(x));
    end
    passive0 = (x==0);
    feax(find(passive0)) = 0.001;
    lcase = eval(get(handles.loadcase,'String'));
    [U]=FEAqfinal(feax, Ymodul, Pratio, pPower, fixedU, fixedF, springK,q);

    [KE]=localK(Ymodul,Pratio);
    for ely = 1:nely
        for elx = 1:nelx
            n1 = (nely+1)*(elx-1)+ely;
            n2 = (nely+1)* elx   +ely;
            Ue1 = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],
1);
            Ue2 = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],
2);
            mpe(ely,elx) = x(ely,elx)^pPower*Ue2'*KE*Ue1;
            se(ely,elx) = 1/2*x(ely,elx)^pPower*Ue1'*KE*Ue1;
        end
    end

    c=mpe;

    % plot MPE
    himg = imagesc(mpe);
    axis equal; axis tight; axis off; colorbar;
    str = [ ' MPEmin: ' sprintf('%6.4f',min(min(mpe))) ...
        ' MPEmax: ' sprintf('%6.4f',max(max(mpe))) ...
        ' MPEtot: ' sprintf('%6.4f',sum(sum(mpe)))];
    title(str);
    disp(str);

    % plot SE
    figure
    himg = imagesc(se);
    axis equal; axis tight; axis off; colorbar;
    str = [ ' SEmin: ' sprintf('%6.4f',min(min(se))) ...
        ' SEmax: ' sprintf('%6.4f',max(max(se))) ...
        ' SEtot: ' sprintf('%6.4f',sum(sum(se)))];
    title(str);
    disp(str);
end
 disp('fea is done')
toc

% --- Executes on button press in closebutton.
function closebutton_Callback(hObject, eventdata, handles)
disp('finishing ndopti')
cl = clock;
hr = [num2str(cl(4)),'h-',num2str(cl(5)),'min-',num2str(cl(6)),'sec'];
str = [date,'   ',hr];
disp(str)
disp('------------------------------------------------------')
```

```
close;


% --- Executes during object creation, after setting all properties.
function sens_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function sens_Callback(hObject, eventdata, handles)


% --- Executes on mouse press over figure background.
function ndoptifig_ButtonDownFcn(hObject, eventdata, handles)
% hObject    handle to ndoptifig (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on button press in actcbx.
function actcbx_Callback(hObject, eventdata, handles)
% hObject    handle to actcbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of actcbx


% --- Executes during object creation, after setting all properties.
function uxedt_CreateFcn(hObject, eventdata, handles)
% hObject    handle to uxedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function uxedt_Callback(hObject, eventdata, handles)
% hObject    handle to uxedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of uxedt as text
%        str2double(get(hObject,'String')) returns contents of uxedt as a double
```

```
% --- Executes during object creation, after setting all properties.
function uyedt_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function uyedt_Callback(hObject, eventdata, handles)


% --- Executes on button press in hcabutton.
function hcabutton_Callback(hObject, eventdata, handles)
ndhcagui


% --- Executes on selection change in springsregion.
function springsregion_Callback(hObject, eventdata, handles)
% hObject    handle to springsregion (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = get(hObject,'String') returns springsregion contents as cell
array
%        contents{get(hObject,'Value')} returns selected item from springsregion


% --- Executes during object creation, after setting all properties.
function springsregion_CreateFcn(hObject, eventdata, handles)
% hObject    handle to springsregion (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function x1springs_Callback(hObject, eventdata, handles)
% hObject    handle to x1springs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of x1springs as text
%        str2double(get(hObject,'String')) returns contents of x1springs as a
double


% --- Executes during object creation, after setting all properties.
```

```
function x1springs_CreateFcn(hObject, eventdata, handles)
% hObject    handle to x1springs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function y1springs_Callback(hObject, eventdata, handles)
% hObject    handle to y1springs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of y1springs as text
%        str2double(get(hObject,'String')) returns contents of y1springs as a
double


% --- Executes during object creation, after setting all properties.
function y1springs_CreateFcn(hObject, eventdata, handles)
% hObject    handle to y1springs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function x2springs_Callback(hObject, eventdata, handles)
% hObject    handle to x2springs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of x2springs as text
%        str2double(get(hObject,'String')) returns contents of x2springs as a
double


% --- Executes during object creation, after setting all properties.
function x2springs_CreateFcn(hObject, eventdata, handles)
% hObject    handle to x2springs (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function y2springs_Callback(hObject, eventdata, handles)
% hObject    handle to y2springs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of y2springs as text
%        str2double(get(hObject,'String')) returns contents of y2springs as a
double


% --- Executes during object creation, after setting all properties.
function y2springs_CreateFcn(hObject, eventdata, handles)
% hObject    handle to y2springs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function kxedt_Callback(hObject, eventdata, handles)
% hObject    handle to kxedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of kxedt as text
%        str2double(get(hObject,'String')) returns contents of kxedt as a double



% --- Executes during object creation, after setting all properties.
function kxedt_CreateFcn(hObject, eventdata, handles)
% hObject    handle to kxedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
```

```
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function kyedt_Callback(hObject, eventdata, handles)
% hObject    handle to kyedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of kyedt as text
%        str2double(get(hObject,'String')) returns contents of kyedt as a double


% --- Executes during object creation, after setting all properties.
function kyedt_CreateFcn(hObject, eventdata, handles)
% hObject    handle to kyedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

% --- Executes on button press in applysprings.
function applysprings_Callback(hObject, eventdata, handles)
global x springK
vspringsregion = get(handles.springsregion,'Value');
vspringkx = eval(get(handles.kxedt,'String'));
vspringky = eval(get(handles.kyedt,'String'));
loadcase = eval(get(handles.loadcase,'String'));
nelx = size(x,2);
nely = size(x,1);
htri = get(handles.resetsprings,'UserData');
x1springs = eval(get(handles.x1springs,'String'));
y1springs = eval(get(handles.y1springs,'String'));
r = (1 - 1/loadcase)/2;
g = (1 - 1/loadcase)/2;
b = (1 - 1/loadcase)/2;
mycolor = [r g 0];
if vspringsregion == 1 %individual node
    if vspringkx %xdir
        springK = [springK; 2*(y1springs+1) + 2*x1springs*(nely+1) - 1,
vspringkx, loadcase];
        if   vspringkx>0
            htri = [htri; plot_spring(x1springs, y1springs, 1, mycolor)];
        else if vspringkx<0
```

```
                htri = [htri; plot_spring(x1springs, y1springs, 2, mycolor)];
            end
        end
    end
    if vspringky %ydir
        springK = [springK; 2*(y1springs+1) + 2*x1springs*(nely+1), vspringky,
loadcase];
        if   vspringky>0
            htri = [htri; plot_spring(x1springs, y1springs, 3, mycolor)];
        else if vspringky<0
                htri = [htri; plot_spring(x1springs, y1springs, 4, mycolor)];
            end
        end
    end
elseif vspringsregion == 2 %multiple nodes
    x2springs = eval(get(handles.x2springs,'String'));
    y2springs = eval(get(handles.y2springs,'String'));
    xmaxrec = max(x1springs,x2springs);
    xminrec = min(x1springs,x2springs);
    ymaxrec = max(y1springs,y2springs);
    yminrec = min(y1springs,y2springs);
    for nodex = xminrec:xmaxrec
        for nodey = yminrec:ymaxrec
            if vspringkx %xdir
                springK = [springK; 2*(nodey+1) + 2*nodex*(nely+1) - 1,
vspringkx, loadcase];
                if   vspringkx>0
                    htri = [htri; plot_spring(nodex, nodey, 1, mycolor)];
                else if vspringkx<0
                        htri = [htri; plot_spring(nodex, nodey, 2, mycolor)];
                    end
                end
            end
            if vspringky %ydir
                springK = [springK; 2*(nodey+1) + 2*nodex*(nely+1), vspringky,
loadcase];
                if   vspringky>0
                    htri = [htri; plot_spring(nodex, nodey, 3, mycolor)];
                else if vspringky<0
                        htri = [htri; plot_spring(nodex, nodey, 4, mycolor)];
                    end
                end
            end
        end
    end
end
% sort column mistake
% %%%Why is this here - commented out to see what would happen
% if(size(springK,1)>1)
%     springK=sort(springK);%i added a comma 1, then deleted it, then added it
% end
% updatefem.springs=1;
% set(handles.resetsprings,'UserData',htri);


% --- Executes on button press in checksprings.
```

```
function checksprings_Callback(hObject, eventdata, handles)
global springK
disp('checking springs...')
springK


% --- Executes on button press in resetsprings.
function resetsprings_Callback(hObject, eventdata, handles)
global springK
harrow = get(handles.resetsprings,'UserData');
delete(harrow);
set(handles.resetsprings,'UserData',[]);
springK = [];
updatefem.springs=1;

% --- Executes on button press in picksprings.
function picksprings_Callback(hObject, eventdata, handles)
vspringsregion = get(handles.springsregion,'Value');
if vspringsregion==1
    coord = ginput(1);
    set(handles.x1springs,'String',round(coord(1,1)));
    set(handles.y1springs,'String',round(coord(1,2)));
elseif vspringsregion==2
    coord = ginput(1);
    set(handles.x1springs,'String',round(coord(1,1)));
    set(handles.y1springs,'String',round(coord(1,2)));
    coord = ginput(1);
    set(handles.x2springs,'String',round(coord(1,1)));
    set(handles.y2springs,'String',round(coord(1,2)));
end



% --- Executes on button press in mmabutton.
function mmabutton_Callback(hObject, eventdata, handles)

topMMAgui;
```

7.2 – Appendix B: Ndhca, Ndhcam, and FEA Functions altered for Support Optimization

******NOTE: Code written by the author in this section is marked by a "%*" on the right side of the page.******

NDHCA.m CODE

```
function newx = ndhca(x, fixedF, fixedU, Kf, E, nu, penal, cset, gMf, k, backt,
neighbors, bcond, scond, itmax, convergence_type, vtol, movflg, movstr,q)

% NDHCA Hybrid Cellular Automata Local Control
%
% Andres Tovar - Univ. of Notre Dame
% last modification: 4-03-06
```

```
% note: ctol was changed for ctol in the input

%Modifications by David lettieri on 2/11/07 marked with %* on right hand
%side

gtol=5e-4;
zpenal = penal;                                                          %*
total_elts=prod(size(x));                                                %*
gKf = sum(sum(sum(q)))/total_elts;                                       %*

% variable introduction
cero = 0;
uno = 0;
xcompliance = 0;
noncompliant = 0;

% INPUT PARAMETERS
[nely, nelx] = size(x);
lcases = max(fixedF(:,3));

% MOVIE
if movflg
    mov = avifile(movstr);
end

% IDENTIFICATION OF PASSIVE AND ACTIVE ELEMENTS
passive0 = (x==0);
passive1 = (x==1);
passiveq0 = (q==0);                                                      %*
x(find(passive0)) = 0.001;
x(find(passive1)) = 1.000;

% START HCA ALGORITHM
warning off MATLAB:nearlySingularMatrixUMFPACK
disp('running ndhca...')
figure; colormap(gray);
title('running ndhca...'); axis off; pause(1e-6);
% intial strain energy c

[U]=FEAqfinal(x, E, nu, penal, fixedU, fixedF, Kf,q);
c = zeros(nely,nelx);
it = 0;

[KE]=localK(E,nu);
for ely = 1:nely
    for elx = 1:nelx
        n1 = (nely+1)*(elx-1)+ely;
        n2 = (nely+1)* elx   +ely;
        for lc=1:lcases
            Ue = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],lc);
            c(ely,elx) = c(ely,elx)+x(ely,elx)^penal*Ue'*KE*Ue;
        end
    end
end
```

```
% plot volume fraction
str=[' It.: ' sprintf('%4i',it) ' Obj.: ' sprintf('%10.4f',sum(sum(c))) ...
    ' Vol.: ' sprintf('%6.3f',sum(sum(x))/(nelx*nely))];
disp(str);
% PLOT DENSITIES
colormap(gray); imagesc(-x,[-1 0]); axis equal; axis tight; axis off;
title(str); pause(1e-6);
% add to the movie
if movflg
    f = getframe(gca);
    mov = addframe(mov,f);
end

% % bounary correction for neighborhoods
if(~scond & bcond==1)
    gneighbors=neighbors;
    neighbors = floor(gneighbors*doavgf(~passive0,gneighbors));
end

% cavgn contains the average state in neighborhoods
if bcond==1; %fixed bcond
    cavgn = doavgf(c,neighbors);
    if(scond) xavgn = doavgf(x,neighbors); end
end

if bcond==2;  %periodic bcond
    cavgn = doavgp(c,neighbors);
    if(scond) xavgn = doavgp(x,neighbors); end
end

% CREATE HISTORY OF STATES
cavgnt = zeros(size(cavgn,1),size(cavgn,2),backt+1);
for t=1:backt+1
    % cavgnt stores cavgn as many times as backt+1
    cavgnt(:,:,t) = cavgn;
end
% scavgn is the sum of all stored cavgn to be used in integral control
scavgn = cavgn*(backt+1);
% oldcavgn is a initially zero;
oldcavgn = zeros(size(cavgn));

% ITERATIONS
change = inf;
newx=zeros(size(x)); deltax=newx; newq = newx; deltaq = newx;                %*
while (it<itmax & change>vtol)
    it = it + 1;

    if(scond)
        cid = xavgn > 0.001 & xavgn < 0.999;
    else
        cid=x~=0;
    end

    total_elts=prod(size(x));
    % Determine which set point to use
```

```
    if(cset)
        setpoint=cset;  % User specified, constant
    else
        setpoint=ones(size(x))*(sum(sum(c)))/prod(size(x));
    end

    setpointq=ones(size(x))*(sum(sum(c)))/prod(size(x));                    %*
    sp(it) = mean(mean(setpoint));
    spq(it) = mean(mean(setpointq));                                        %*
    loop=1;
    loopq=1;                                                                %*
    while loop || loopq                                                     %*

        % CA RULE -> newx
        if(k(1)~=0)   % apply  fixed control
            newx(cid) = x(cid) + k(1)*sign(cavgn(cid)-setpoint(cid));
            newq(cid) = q(cid) + k(1)*sign(cavgn(cid)-setpointq(cid));      %*
        else if (k(5)~=0)   % apply  ratio technque
                newx(cid) = k(5)*x(cid).*(cavgn(cid)./setpoint(cid)).^(1/penal);
                newq(cid) =
k(5)*q(cid).*(cavgn(cid)./setpointq(cid)).^(1/zpenal);        %*
            else if (k(6)~=0)  % apply  ratio technque
                    newx(cid) =
k(6)*x(cid).*(setpoint(cid)./cavgn(cid)).^(1/penal-1);
                    newq(cid) =
k(6)*q(cid).*(setpointq(cid)./cavgn(cid)).^(1/zpenal-1);  %*
                else    % apply  PID control

                    %relative mass change
                    deltax(cid) = k(2)*((cavgn(cid)-setpoint(cid))./
setpoint(cid)); % proportional error
                    deltax(cid)  = deltax(cid)+k(3)*((scavgn(cid) -
(backt+1)*setpoint(cid))./ setpoint(cid));% integral error
                    deltax(cid) = deltax(cid)+ k(4)*((cavgn(cid) -
oldcavgn(cid))./ setpoint(cid));  % derivative error

                    %relative stiffness change
                    deltaq(cid) = k(2)*((cavgn(cid)-setpointq(cid))./
setpointq(cid)); % proportional error                                  %*
                    deltaq(cid)  = deltaq(cid)+k(3)*((scavgn(cid) -
(backt+1)*setpointq(cid))./ setpointq(cid));% integral error      %*
                    deltaq(cid) = deltaq(cid)+ k(4)*((cavgn(cid) -
oldcavgn(cid))./ setpointq(cid));  % derivative error              %*

                    % Apply denisty change limits
                    maxchange=0.25;

                    %relative mass change limits
                    deltax(deltax(cid) > 0) = min(maxchange, deltax(deltax(cid)
> 0));
                    deltax(deltax(cid) < 0) = max(-maxchange, deltax(deltax(cid)
< 0));
                    newx(cid) = x(cid) + deltax(cid);

                    %relative stiffness change limits
```

```
                    deltaq(deltaq(cid) > 0) = min(maxchange, deltaq(deltaq(cid)
> 0));      %*
                    deltaq(deltaq(cid) < 0) = max(-maxchange, deltaq(deltaq(cid)
< 0));     %*
                    newq(cid) = q(cid) + deltaq(cid);
%*
                end
            end
        end

        %check density saturation
        %relative mass
        newx0 = (newx < 0.001); newx(find(newx0)) = 0.001;
        newx1 = (newx > 0.999); newx(find(newx1)) = 0.999;
        %relative stiffness
        newq0 = (newq < 0.001); newq(find(newq0)) = 0.001;                  %*
        newq1 = (newq > 0.999); newq(find(newq1)) = 0.999;                  %*

        % force density value in passive x
        newx(find(passive0)) = 0.001;
        newx(find(passive1)) = 1.000;

        newq(find(passiveq0)) = 0;                                          %*
        Mf=sum(sum(sum(newx)))/total_elts;
        Kfa=sum(sum(sum(newq)))/total_elts;                                 %*

        if(gMf);setpoint=setpoint+setpoint*(Mf-gMf)/gMf; else loop=0;end;  %*
        loop=abs(gMf-Mf)>gtol & min(min(setpoint(cid))>1e-10); % if mass is not
within tolerance, set flag to iterate
        setpointq=setpointq+setpointq*(Kfa-gKf)/gKf;                        %*
        loopq=abs(gKf-Kfa)>gtol & min(min(setpointq(cid))>1e-10); % if stiffness
is not within tolerance, set flag to iterate
    end


    %     lowerbound=(newx==0.001); % leave void  elements uncorrected
    %     upperbound=(newx==0.999); % leave fully dense elements uncorrected
    %
    %     if(gMf)
    %         i=0;
    %         while(abs(Mf-gMf)>=gtol & i < 100)             % Compare current
Mf to constraint; iterate until converged or max 100 iterations
    %             if (gMf-Mf)>0
    %                 newx(~lowerbound)=newx(~lowerbound)*(gMf/Mf);
    %             else
    %                 newx(~upperbound)=newx(~upperbound)*(gMf/Mf);
    %             end
    %
    %             %newx=newx*(gMf/Mf);
    %
    %             i=i+1;
    %
    %             newx0 = (newx < 0.001); newx(find(newx0)) = 0.001;
    %             newx1 = (newx > 0.999); newx(find(newx1)) = 0.999;
    %             newx(passive0) = 0.001;
    %             newx(passive1) = 1;
```

```
    %
    %             Mf=sum(sum(sum(newx)))/total_elts;
    %         end
    %     end

    %check numer of cels not in compliance with convergence criteria
    xcompliance=abs(newx-x);
    %     noncompliant = 0;
    %     for cero = 1:nely
    %         for uno = 1:nelx
    %             if xcompliance(cero,uno) > vtol
    %                 noncompliant = noncompliant + 1;
    %             else
    %                 noncompliant = noncompliant;
    %             end
    %         end
    %     end
    %     noncompliant
    % function call - strain energy newc
    [U]=FEAqfinal(newx, E, nu, penal, fixedU, fixedF, Kf,q);
    newc = sparse(nely,nelx);
    for ely = 1:nely
        for elx = 1:nelx
            n1 = (nely+1)*(elx-1)+ely;
            n2 = (nely+1)* elx   +ely;
            for lcase=1:lcases
                Ue = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],
lcase);
                newc(ely,elx) = newc(ely,elx) + newx(ely,elx)^penal*Ue'*KE*Ue;
            end
        end
    end

    % convergence criteria
    if(convergence_type==1)
        change = sum(sum(abs(newx-x)));        %Global convergence criteria
    else
        change = max(max(abs(newx-x)));       %Local convergence criteria
    end

    % update state of x
    x = full(newx);
    c = full(newc);
    q = full(newq);                                                          %*
    % store old cavgn to be used by derivative control
    oldcavgn = cavgn;

    % cavgn contains the average state in neighborhoods
    if bcond==1; %fixed bcond
        cavgn = doavgf(c,neighbors);
        if(scond) xavgn = doavgf(x,neighbors); end
    end

    if bcond==2;  %periodic bcond
        cavgn = doavgp(c,neighbors);
        if(scond) xavgn = doavgp(x,neighbors); end
```

```
    end

    % sum of stored states
    scavgn = scavgn + cavgn - cavgnt(:,:,1);
    cavgnt(:,:,backt+2) = cavgn;
    for t=1:backt+1
        cavgnt(:,:,t) = cavgnt(:,:,t+1);
    end

    % plot
    str=[' It.: ' sprintf('%4i',it) ' Obj.: ' sprintf('%10.4f',sum(sum(c))) ...
        ' Vol.: ' sprintf('%6.3f',sum(sum(x))/(nelx*nely)) ...
        ' ch.: ' sprintf('%6.3f',change )];
    disp(str);
    % PLOT DENSITIES
    colormap(gray); imagesc(-x,[-1 0]); axis equal; axis tight; axis off;
title(str); pause(1e-6);
    if movflg
        f = getframe(gca);
        mov = addframe(mov,f);
    end
end

% close movie
if movflg
    mov = close(mov);
end
figure
plot(sp,'o')
hold on                                                                 %*
plot(spq,'r.')                                                          %*
hold off                                                                %*
title('Setpoint Each Iteration')
xlabel('Iteration')
ylabel('Setpoint')
legend('Mass setpoint','Stiffness Setpoint')                            %*
disp('ndhca is done')
```

NCHCAM.m CODE

```
function newx = ndhca(x, extF, fixedU, Kf, E, nu, penal, cset, gMf, k, backt,
neighbors, bcond, scond, itmax, convergence_type, vtol, movflg, movstr,q)
% NDHCA Hybrid Cellular Automata Local Control
%
% Andres Tovar - Univ. of Notre Dame
% last modification: 4-03-06
% note: ctol was changed for ctol in the input

gtol=5e-4;
zpenal = penal+1;                                                          %*
total_eltsq = prod(size(find(q>0)));                                       %*
gKf = sum(sum(sum(q)))/total_eltsq;                                        %*

% variable introduction
cero = 0;
uno = 0;
xcompliance = 0;
noncompliant = 0;

% INPUT PARAMETERS
[nely, nelx] = size(x);
lcases = max(extF(:,3));

% MOVIE
if movflg
    mov = avifile(movstr);
end

% IDENTIFICATION OF PASSIVE AND ACTIVE ELEMENTS
passive0 = (x==0);
passive1 = (x==1);
passiveq0 = (q==0);                                                        %*
x(find(passive0)) = 0.001;
x(find(passive1)) = 1.000;

% START HCA ALGORITHM
warning off MATLAB:nearlySingularMatrixUMFPACK
disp('running ndhca...')
figure; colormap(gray);
title('running ndhca...'); axis off; pause(1e-6);

% intial mutual potential energy, c, and strain energy, cq
[U, Uout]=FEAqfinal(x, E, nu, penal, fixedU, extF, Kf,q);
c = zeros(nely,nelx);           %mutual potential energy
cq = zeros(nely,nelx);          %strain energy                             %*
it = 0;
[KE]=localK(E,nu);

%mutual potential energy                                                   %*
for ely = 1:nely
    for elx = 1:nelx
        n1 = (nely+1)*(elx-1)+ely;
        n2 = (nely+1)* elx   +ely;
        Ue1 = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],1);
```

```
        Ue2 = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],2);
        c(ely,elx) = c(ely,elx) + x(ely,elx)^penal*Ue1'*KE*Ue2;
%CHECK THIS SIGN!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    end
end
c=-c;

%strain energy density for attachment                                          %*
for ely = 1:nely                                                               %*
    for elx = 1:nelx                                                           %*
        n1 = (nely+1)*(elx-1)+ely;                                             %*
        n2 = (nely+1)* elx   +ely;                                             %*
        for lc=1:lcases                                                        %*
            Ue = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],lc);
%*
            cq(ely,elx) = cq(ely,elx)+x(ely,elx)^penal*Ue'*KE*Ue;              %*
        end                                                                    %*
    end                                                                        %*
end                                                                            %*


% plot volume fraction
str=[' It.: ' sprintf('%4i',it) ' Obj.: ' sprintf('%10.4f',Uout(1)) ...
    ' Vol.: ' sprintf('%6.3f',sum(sum(x))/(nelx*nely))];
disp(str);
% PLOT DENSITIES
colormap(gray); imagesc(-x,[-1 0]); axis equal; axis tight; axis off;
title(str); pause(1e-6);

% add to the movie
if movflg
    f = getframe(gca);
    mov = addframe(mov,f);
end

% % bounary correction for neighborhoods
if(~scond & bcond==1)
    gneighbors=neighbors;
    neighbors = floor(gneighbors*doavgf(~passive0,gneighbors));
end

% cavgn contains the average state in neighborhoods
if bcond==1; %fixed bcond
    cavgn = doavgf(c,neighbors);
    cavgnq = doavgf(cq,neighbors);                                             %*
Does not account for surface boundary conditions
    if(scond) xavgn = doavgf(x,neighbors); end
end

if bcond==2;  %periodic bcond
    cavgn = doavgp(c,neighbors);
    cavgnq = doavgf(cq,neighbors);                                             %*
Does not account for surface boundary conditions
    if(scond) xavgn = doavgp(x,neighbors); end
end
```

```
cavgnq(find(passiveq0)) = 0;                                                %*
So that setpoint average does not count cells that are passive

% CREATE HISTORY OF STATES
cavgnt = zeros(size(cavgn,1),size(cavgn,2),backt+1);
cavgntq = zeros(size(cavgnq,1),size(cavgnq,2),backt+1);                     %*
for t=1:backt+1
    % cavgnt stores cavgn as many times as backt+1
    cavgnt(:,:,t) = cavgn;
    cavgntq(:,:,t) = cavgnq;                                               %*
end

% scavgn is the sum of all stored cavgn to be used in integral control
scavgn = cavgn*(backt+1);
scavgnq = cavgnq*(backt+1);                                                 %*

% oldcavgn is a initially zero;
oldcavgn = zeros(size(cavgn));
oldcavgnq = zeros(size(cavgnq));                                            %*

% ITERATIONS
change = inf;
newx=zeros(size(x)); deltax=newx; newq = newx; deltaq = newx;               %*
while (it<itmax & change>vtol)
    it = it + 1;

    if(scond)      %I have not implemented surface boundary conditions
        cid = xavgn > 0.001 & xavgn < 0.999;
    else
        cid=x~=0;
        cidq=q~=0;                                                          %*
    end

    total_elts=prod(size(x));
    % Determine which set point to use
    if(cset)
        setpoint=cset;  % User specified, constant
    else
        ctavg=ones(size(x))*abs(sum(sum(c)))/prod(size(x));
        setpoint=ctavg;
    end

    ctavgq = ones(size(x))*abs(sum(sum(cq)))/total_eltsq;                   %*
    setpointq=ctavgq;                                                       %*
    sp(it) = mean(mean(setpoint));                                          %*
    spq(it) = mean(mean(setpointq));                                        %*

    if(gMf)
        stop=0; stopq = 0;                                                  %*
        Mf=0; Kfa = 0;                                                      %*
        while abs(gMf-Mf)>gtol & abs(gKf-Kfa)>gtol & ~stop &~stopq          %*
            % CA RULE -> newx
            if(k(1)~=0)   % apply  fixed control
                newx(cid) = x(cid) + k(1)*sign(cavgn(cid)-setpoint(cid));
```

```
                newq(cid) = q(cid) + k(1)*sign(cavgnq(cid)-setpointq(cid)); %*
            else if (k(5)~=0)   % apply  ratio technque
                    %newx(cid) =
k(5)*x(cid).*(cavgn(cid)./setpoint(cid)).^(1/penal);
                    deltax(cid) =
k(5)*x(cid).*sign(cavgn(cid)./setpoint(cid)).*abs(cavgn(cid)./setpoint(cid)).^(1
/(penal));
                    deltaq(cid) =
k(5)*q(cid).*sign(cavgnq(cid)./setpointq(cid)).*abs(cavgnq(cid)./setpointq(cid))
.^(1/(zpenal));    %*
                    % Apply denisty change limits
                    maxchange=0.1;
                    newx(deltax(cid) > 0) = min(maxchange, deltax(deltax(cid) >
0));
                    newq(deltaq(cid) > 0) = min(maxchange, deltaq(deltaq(cid) >
0));        %*
                    newx(deltax(cid) < 0) = max(-maxchange, deltax(deltax(cid) <
0));
                    newq(deltaq(cid) < 0) = max(-maxchange, deltaq(deltaq(cid) <
0));       %*
                else if (k(6)~=0)  % apply  ratio technque
                        newx(cid) =
k(6)*x(cid).*(setpoint(cid)./cavgn(cid)).^(1/penal-1);
                        newq(cid) =
k(6)*q(cid).*(setpointq(cid)./cavgnq(cid)).^(1/zpenal-1);  %*
                    else    % apply  PID control

                        %relative mass change
                        deltax(cid) = k(2)*((cavgn(cid)-setpoint(cid))./
setpoint(cid)); % proportional error
                        deltax(cid)  = deltax(cid)+k(3)*((scavgn(cid) -
(backt+1)*setpoint(cid))./ setpoint(cid));% integral error
                        deltax(cid) = deltax(cid)+ k(4)*((cavgn(cid) -
oldcavgn(cid))./ setpoint(cid));  % derivative error

                        %relative stiffness change
                        deltaq(cidq) = k(2)*((cavgnq(cidq)-setpointq(cidq))./
setpointq(cidq)); % proportional error                              %*
                        deltaq(cidq)  = deltaq(cidq)+k(3)*((scavgnq(cidq) -
(backt+1)*setpointq(cidq))./ setpointq(cidq));% integral error      %*
                        deltaq(cidq) = deltaq(cidq)+ k(4)*((cavgnq(cidq) -
oldcavgnq(cidq))./ setpointq(cidq));  % derivative error              %*

                        % Apply denisty change limits
                        maxchange=0.1;

                        %relative mass change limits
                        deltax(deltax(cid) > 0) = min(maxchange,
deltax(deltax(cid) > 0));
                        deltax(deltax(cid) < 0) = max(-maxchange,
deltax(deltax(cid) < 0));
                        newx(cid) = x(cid) + deltax(cid);

                        %relative stiffness change limits
                        deltaq(deltaq(cidq) > 0) = min(maxchange,
deltaq(deltaq(cidq) > 0));      %*
```

```
                        deltaq(deltaq(cidq) < 0) = max(-maxchange,
deltaq(deltaq(cidq) < 0));     %*

                        %applying scaling
                        scaledelmts = find(q>0);
%*
                        scaledelmts =
scaledelmts(length(scaledelmts)/2+1:length(scaledelmts));   %*
                        scalefactor = 10;
                        deltaq(scaledelmts) = deltaq(scaledelmts)/scalefactor;
%*
                        newq(cidq) = q(cidq) + deltaq(cidq);
%*
                    end
                end
            end

            %check density saturation
            %relative mass
            newx0 = (newx < 0.001); newx(find(newx0)) = 0.001;
            newx1 = (newx > 0.999); newx(find(newx1)) = 0.999;
            %relative stiffness
            newq0 = (newq < 0.0001); newq(find(newq0)) = 0.0001;           %*
            newq1 = (newq > 0.999); newq(find(newq1)) = 0.999;             %*

            % force density value in passive x
            newx(find(passive0)) = 0.001;
            newx(find(passive1)) = 1.000;

            newq(find(passiveq0)) = 0;                                     %*
            Mf=sum(sum(sum(newx)))/total_elts;
            Kfa=sum(sum(sum(newq)))/total_eltsq;                           %*

            setpoint=setpoint+setpoint*(Mf-gMf)/gMf;
            setpointq=setpointq+setpointq*(Kfa-gKf)/gKf;                   %*

            cavgnqstorage = cavgnq(cavgnq~=0);
%*necessary so not dividing by zero 2 lines below
            stop=~gMf | abs(min(min(setpoint))/min(min(cavgn)))<1e-10;
            stopq=~gKf | abs(min(min(setpointq))/min(min(cavgnqstorage)))<1e-10;
        end

    else
        % CA RULE -> newx
        if(k(1)~=0)   % apply  fixed control
            newx(cid) = x(cid) + k(1)*sign(cavgn(cid)-setpoint(cid));
            newq(cid) = q(cid) + k(1)*sign(cavgnq(cid)-setpointq(cid));    %*
        else if (k(5)~=0)   % apply  ratio technque
                newx(cid) = k(5)*x(cid).*(cavgn(cid)./setpoint(cid)).^(1/penal);
                newq(cid) =
k(5)*q(cid).*(cavgnq(cid)./setpointq(cid)).^(1/zpenal);        %*
            else if (k(6)~=0)  % apply  ratio technque
                    newx(cid) =
k(6)*x(cid).*(setpoint(cid)./cavgn(cid)).^(1/penal-1);
```

```
                    newq(cid) =
k(6)*q(cid).*(setpointq(cid)./cavgnq(cid)).^(1/zpenal-1);  %*
                else    % apply  PID control
                    %relative mass change
                    deltax(cid) = k(2)*((cavgn(cid)-setpoint(cid))./
setpoint(cid)); % proportional error
                    deltax(cid)  = deltax(cid)+k(3)*((scavgn(cid) -
(backt+1)*setpoint(cid))./ setpoint(cid));% integral error
                    deltax(cid) = deltax(cid)+ k(4)*((cavgn(cid) -
oldcavgn(cid))./ setpoint(cid));  % derivative error

                    %relative stiffness change
                    deltaq(cidq) = k(2)*((cavgnq(cidq)-setpointq(cidq))./
setpointq(cidq)); % proportional error                          %*
                    deltaq(cidq)  = deltaq(cidq)+k(3)*((scavgnq(cidq) -
(backt+1)*setpointq(cidq))./ setpointq(cidq));% integral error      %*
                    deltaq(cidq) = deltaq(cidq)+ k(4)*((cavgnq(cidq) -
oldcavgnq(cidq))./ setpointq(cidq));  % derivative error              %*

                    % Apply denisty change limits
                    maxchange=0.2;

                    %relative mass change limits
                    deltax(deltax(cid) > 0) = min(maxchange, deltax(deltax(cid)
> 0));
                    deltax(deltax(cid) < 0) = max(-maxchange, deltax(deltax(cid)
< 0));
                    newx(cid) = x(cid) + deltax(cid);

                    %relative stiffness change limits
                    deltaq(deltaq(cidq) > 0) = min(maxchange,
deltaq(deltaq(cidq) > 0));      %*
                    deltaq(deltaq(cidq) < 0) = max(-maxchange,
deltaq(deltaq(cidq) < 0));     %*
                    newq(cidq) = q(cidq) + deltaq(cidq);
%*

                end
            end
        end
    end

    %check density saturation
    %relative mass
    newx0 = (newx < 0.001); newx(find(newx0)) = 0.001;
    newx1 = (newx > 0.999); newx(find(newx1)) = 0.999;
    %relative stiffness
    newq0 = (newq < 0.0001); newq(find(newq0)) = 0.0001;                    %*
    newq1 = (newq > 0.999); newq(find(newq1)) = 0.999;                      %*

    % force density value in passive x
    newx(find(passive0)) = 0.001;
    newx(find(passive1)) = 1.000;

    % force stiffness value in passive q
```

```
    newq(find(passiveq0)) = 0;                                              %*
    Mf=sum(sum(sum(newx)))/total_elts;
    Kfa=sum(sum(sum(newq)))/total_eltsq;                                    %*

    %     lowerbound=(newx==0.001); % leave void  elements uncorrected
    %     upperbound=(newx==0.999); % leave fully dense elements uncorrected
    %
    %     if(gMf)
    %         i=0;
    %         while(abs(Mf-gMf)>=gtol & i < 100)             % Compare current
Mf to constraint; iterate until converged or max 100 iterations
    %             if (gMf-Mf)>0
    %                 newx(~lowerbound)=newx(~lowerbound)*(gMf/Mf);
    %             else
    %                 newx(~upperbound)=newx(~upperbound)*(gMf/Mf);
    %             end
    %
    %             %newx=newx*(gMf/Mf);
    %
    %             i=i+1;
    %
    %             newx0 = (newx < 0.001); newx(find(newx0)) = 0.001;
    %             newx1 = (newx > 0.999); newx(find(newx1)) = 0.999;
    %             newx(passive0) = 0.001;
    %             newx(passive1) = 1;
    %
    %             Mf=sum(sum(sum(newx)))/total_elts;
    %         end
    %     end

    %check numer of cels not in compliance with convergence criteria
    xcompliance=abs(newx-x);
    %     noncompliant = 0;
    %     for cero = 1:nely
    %         for uno = 1:nelx
    %             if xcompliance(cero,uno) > vtol
    %                 noncompliant = noncompliant + 1;
    %             else
    %                 noncompliant = noncompliant;
    %             end
    %         end
    %     end
    %     noncompliant
    % function call - strain energy newc
    [U, Uout]=FEAqfinal(newx, E, nu, penal, fixedU, extF, Kf,q);
    %mutual potential energy
    c = zeros(nely,nelx);
    for ely = 1:nely
        for elx = 1:nelx
            n1 = (nely+1)*(elx-1)+ely;
            n2 = (nely+1)* elx   +ely;
            dc(ely,elx) = 0.;
            Ue1 = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],
1);
            Ue2 = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],
2);
```

```
            c(ely,elx) = c(ely,elx) + x(ely,elx)^penal*Ue2'*KE*Ue1;
        end
    end
    c=-c;

    %strain energy density for attachment
%*
    for ely = 1:nely
%*
        for elx = 1:nelx
%*
            n1 = (nely+1)*(elx-1)+ely;
%*
            n2 = (nely+1)* elx   +ely;
%*
            for lc=1:lcases
%*
                Ue = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2;
2*n1+1;2*n1+2],lc); %*
                cq(ely,elx) = cq(ely,elx)+x(ely,elx)^penal*Ue'*KE*Ue;
%*
            end
%*
        end
%*
    end
%*

    cavgnq(find(passiveq0)) = 0;
%* So that setpoint average does not count cells that are passive

    % convergence criteria
    if(convergence_type==1)
        change = sum(sum(abs(newx-x)));        %Global convergence criteria
    else
        change = max(max(abs(newx-x)));       %Local convergence criteria
    end

    % update state of x
    x = full(newx);
    q = full(newq);                                                    %*

    % store old cavgn to be used by derivative control
    oldcavgn = cavgn;
    oldcavgnq = cavgnq;                                                %*

    % cavgn contains the average state in neighborhoods
    if bcond==1; %fixed bcond
        cavgn = doavgf(c,neighbors);
        cavgnq = doavgf(cq,neighbors);                                 %*
        if(scond) xavgn = doavgf(x,neighbors); end    %I dod not account for
surface conditions
    end

    if bcond==2;  %periodic bcond
```

```
        cavgn = doavgp(c,neighbors);
        cavgnq = doavgp(cq,neighbors);                                    %*
        if(scond) xavgn = doavgp(x,neighbors); end    %I dod not account for
surface conditions
    end

    % sum of stored states
    scavgn = scavgn + cavgn - cavgnt(:,:,1);
    scavgnq = scavgnq + cavgnq - cavgntq(:,:,1);                          %*

    cavgnt(:,:,backt+2) = cavgn;
    cavgntq(:,:,backt+2) = cavgnq;                                        %*
    for t=1:backt+1
        cavgnt(:,:,t) = cavgnt(:,:,t+1);
        cavgntq(:,:,t) = cavgntq(:,:,t+1);                                %*
    end

    % plot
    str=[' It.: ' sprintf('%4i',it) ' Obj.: ' sprintf('%10.4f',sum(Uout(1))) ...
        ' Vol.: ' sprintf('%6.3f',sum(sum(x))/(nelx*nely)) ...
        ' ch.: ' sprintf('%6.3f',change )];
    disp(str);
    % PLOT DENSITIES
    %colormap(gray); imagesc(-x,[-1 0]); title(str); pause(1e-6);
    colormap(gray); imagesc(-x,[-1 0]); axis equal; axis tight; axis off;
title(str); pause(1e-6);
    if movflg
        f = getframe(gca);
        mov = addframe(mov,f);
    end
end

% close movie
if movflg
    mov = close(mov);
end
figure                                                                    %*
plot(sp,'o')                                                              %*
hold on                                                                   %*
plot(spq,'r.')                                                            %*
hold off                                                                  %*
title('Setpoint Each Iteration')                                          %*
xlabel('Iteration')                                                       %*
ylabel('Setpoint')                                                        %*
legend('Mass setpoint','Stiffness Setpoint')                              %*
disp('ndhca is done')                                                     %*
save q q
save x x
save Kfa Kfa
figure                                                                    %*
title('Relative Stiffness (q) grayscale')
colormap(gray); imagesc(-q,[-1 0]); axis equal; axis tight; axis off;
title(str); %*
```

FINITE ELEMENT ANALYSIS CODE

```
%%%%%%%%%% FE-ANALYSIS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [U, Uout]=FEAq(x, E, nu, penal, fixedU, extF, Kf,q)
global Kfa
% Returns the displacement vector U and displacement at output (dout) for
compliant mechanism synthesis


zpenal = penal+1;                                                          %*
Ko = 10^10;                                                               %*
[nely,nelx]=size(x);
lcase = max(extF(:,3));

[KE]=localK(E,nu);

% find node for the output displacement for compliant mechanism synthesis
dout=extF(find(extF(:,3)==2), 1);

% Initialize global stiffness matrix, and force and displacement vectors
K = sparse(2*(nelx+1)*(nely+1), 2*(nelx+1)*(nely+1));
F = sparse(2*(nely+1)*(nelx+1), lcase);
U = sparse(2*(nely+1)*(nelx+1), lcase);

% Obtain global stiffness matrix
for elx = 1:nelx
  for ely = 1:nely
    n1 = (nely+1)*(elx-1)+ely;
    n2 = (nely+1)* elx   +ely;
    edof = [2*n1-1; 2*n1; 2*n2-1; 2*n2; 2*n2+1; 2*n2+2; 2*n1+1; 2*n1+2];
    K(edof,edof) = K(edof,edof) + x(ely,elx)^penal*KE;
  end
end



% DEFINE LOADS AND SUPPORTS
% loads
for i = 1:size(extF,1)
    F(extF(i,1),extF(i,3)) = F(extF(i,1),extF(i,3)) + extF(i,2);
end

% Adding in the external spring stiffnesses (K+Kf)
for i = 1:size(Kf,1)
    %if(Kf(i,3)==j)
        K(Kf(i,1),Kf(i,1))=K(Kf(i,1),Kf(i,1))+Kf(i,2);
    %end
end




% %%%%%%%%%%%%%%%%%%%%%%%%%%%%THIS IS PROBLEM%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% % Adding in the attachment opimization spring stiffnesses (K+Kf)+(Kfa)     %*
% for elx = 1:nelx                                                          %*
%   for ely = 1:nely                                                        %*
%     n1 = (nely+1)*(elx-1)+ely;                                            %*
```

```
%     n2 = (nely+1)* elx   +ely;                                                %*
%     edof = [2*n1-1; 2*n1; 2*n2-1; 2*n2; 2*n2+1; 2*n2+2; 2*n1+1; 2*n1+2];   %*
%     K(edof,edof) = K(edof,edof) + q(ely,elx)^zpenal*Ko;                       %*
%   end
% end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%THIS IS PROBLEM%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Adding in the attachment opimization spring stiffnesses (K+Kf)+(Kfa)     %*
for elx = 1:nelx                                                           %*
  for ely = 1:nely                                                         %*
    n1 = (nely+1)*(elx-1)+ely;                                             %*
    n2 = (nely+1)* elx   +ely;                                             %*
    edof = [2*n1-1; 2*n1; 2*n2-1; 2*n2; 2*n2+1; 2*n2+2; 2*n1+1; 2*n1+2];   %*
    Kfa = [Kfa; [edof] [ones(size(edof))*q(ely,elx)^zpenal*Ko]
[ones(size(edof))]];                    %*
  end
end
% Adding in the attachement spring stiffnesses (K+Kf)+(Kfa)
for i = 1:size(Kfa,1)
    %if(Kf(i,3)==j)
        K(Kfa(i,1),Kfa(i,1))=K(Kfa(i,1),Kfa(i,1))+Kfa(i,2);
    %end
end



% supports
alldofs     = [1:2*(nely+1)*(nelx+1)];

for lc = 1:lcase
    %if fixed disp is not specified for loadcase, default to loadcase 1  %*
    if fixedU                                                            %*
        fixeddofs = fixedU(find(fixedU(:,3)==lc), 1);                    %*
        fixeddis = fixedU(find(fixedU(:,3)==lc), 2);                         %*
        if isempty(fixeddofs)                                                %*
            fixeddofs = fixedU(find(fixedU(:,3)==1), 1);                     %*
            fixeddis = fixedU(find(fixedU(:,3)==1), 2);                      %*
        end                                                                  %*
        freedofs = setdiff(alldofs,fixeddofs);                               %*
    else                                                                     %*
        freedofs = alldofs;                                                   %*
    end                                                                      %*
    %solution
    warning off MATLAB:nearlySingularMatrixUMFPACK
    U(freedofs,lc) = K(freedofs,freedofs) \ F(freedofs,lc);

    if fixedU                                                            %*
        U(fixeddofs,lc) = fixeddis;                                      %*
    end                                                                  %*
end

Uout=U(dout,1); % output displacement for 1st loadcase at the point of loading
in the second loadcase
```

7.3 – Appendix C: Remaining Functions Necessary for Program

******NOTE: None of the code in this appendix was written by the author.******

```
function cavgn = doavgf(c,neighbors)
% DOAVGF(cells, neighbors) sums around each cells using fixed boundary
% conditions

nom_neighbors=max(max(neighbors));

[nely,nelx] = size(c);
if (isscalar(neighbors) &  neighbors == 'all'  ) % Global (nelx*ney-1)
    neighbors = nely*nelx-1;
    csum = ones(size(c))*sum(sum(c));
else
    if(isscalar(neighbors)) neighbors=neighbors*ones(size(c)); end

    csum = c; % no neighbors (N=0)
    if nom_neighbors > 0 % von Newmann (N=4)
        % extends c field for fixed BC
        cext(nely+2, nelx+2) = 0;
        cext(2:nely+1, 2:nelx+1) = c;
        x = 2:nelx+1;
        y = 2:nely+1;
        csum = csum + (cext(y,x+1) + cext(y,x-1) + cext(y+1,x) + cext(y-1,x));
        if nom_neighbors > 4 % Moore (N=8)
            csum = csum + cext(y+1,x+1) + cext(y-1,x-1) + cext(y+1,x-1) +
cext(y-1,x+1);
            if nom_neighbors > 8 % Extended Moore (N=12)
                cextm(nely+4, nelx+4) = 0;
                cextm(3:nely+2, 3:nelx+2) = c;
                y = 3:nely+2; x = 3:nelx+2;
                csum = csum + cextm(y-2, x) + cextm(y+2, x) +cextm(y, x-2) +
cextm(y, x+2);
                if nom_neighbors > 12 % Extended Moore (N=24)
                    csum = csum + ...
                        cextm(y-2, x-2) + cextm(y-2, x-1) + cextm(y-2, x+1) +
cextm(y-2, x+2) +...
                        cextm(y+2, x-2) + cextm(y+2, x-1) + cextm(y+2, x+1) +
cextm(y+2, x+2) +...
                        cextm(y-1, x-2)  + cextm(y+1, x-2) + ...
                        cextm(y-1, x+2) + cextm(y+1, x+2);
                end
            end
        end
    end
end
cavgn = csum./(neighbors+1); % average csum (matrix)


%=========================================================================
function cavg = doavgp(c,neighbors)
% DOAVGP(X, neighbors) sums around each X using fixed periodic
% conditions
%=========================================================================
```

```
[nely,nelx] = size(c);
x = 1:nelx;
y = 1:nely;
if neighbors > 24 % Global (nelx*ney-1)
    csum = ones(size(c))*sum(sum(c));
else
    csum = c; % no neighbors (N=0)
    if neighbors > 0 % von Newmann (N=4)
        csum(mod(y,nely)+1, mod(x,nelx)+1) = csum(mod(y,nely)+1, mod(x,nelx)+1)
+...
            c(mod(y,nely)+1, mod(x+1,nelx)+1) + c(mod(y,nely)+1, mod(x-
1,nelx)+1) +...
            c(mod(y+1,nely)+1, mod(x,nelx)+1) + c(mod(y-1,nely)+1,
mod(x,nelx)+1);
        if neighbors > 4 % Moore (N=8)
            csum(mod(y,nely)+1, mod(x,nelx)+1) = csum(mod(y,nely)+1,
mod(x,nelx)+1) +...
                c(mod(y+1,nely)+1, mod(x+1,nelx)+1) + c(mod(y-1,nely)+1, mod(x-
1,nelx)+1) + ...
                c(mod(y+1,nely)+1, mod(x-1,nelx)+1) + c(mod(y-1,nely)+1,
mod(x+1,nelx)+1);
            if neighbors > 8 % MvonN (N=12)
                csum(mod(y,nely)+1, mod(x,nelx)+1) = csum(mod(y,nely)+1,
mod(x,nelx)+1) +...
                    c(mod(y-2,nely)+1, mod(x,nelx)+1) + c(mod(y+2,nely)+1,
mod(x,nelx)+1) + ...
                    c(mod(y,nely)+1, mod(x-2,nelx)+1) + c(mod(y,nely)+1,
mod(x+2,nelx)+1);
                if neighbors > 12 % Extended Moore (N=24)
                    csum(mod(y,nely)+1, mod(x,nelx)+1) = csum(mod(y,nely)+1,
mod(x,nelx)+1) +...
                        c(mod(y-2,nely)+1, mod(x-2,nelx)+1) + c(mod(y-2,nely)+1,
mod(x-1,nelx)+1) + ...
                        c(mod(y-2,nely)+1, mod(x+1,nelx)+1) + c(mod(y-2,nely)+1,
mod(x+2,nelx)+1) +...
                        c(mod(y+2,nely)+1, mod(x-2,nelx)+1) + c(mod(y+2,nely)+1,
mod(x-1,nelx)+1) + ...
                        c(mod(y+2,nely)+1, mod(x+1,nelx)+1) + c(mod(y+2,nely)+1,
mod(x+2,nelx)+1) +...
                        c(mod(y-1,nely)+1, mod(x-2,nelx)+1) + c(mod(y+1,nely)+1,
mod(x-2,nelx)+1) + ...
                        c(mod(y-1,nely)+1, mod(x+2,nelx)+1) + c(mod(y+1,nely)+1,
mod(x+2,nelx)+1);
                end
            end
        end
    end
end
cavg = csum./(neighbors+1); % average csum (matrix)



% Obtain elemental stiffness matrix
function [KE]=localK(E,nu)

k=[ 1/2-nu/6   1/8+nu/8 -1/4-nu/12 -1/8+3*nu/8 ...
```

```
   -1/4+nu/12 -1/8-nu/8  nu/6       1/8-3*nu/8];
KE = E/(1-nu^2)*[ k(1) k(2) k(3) k(4) k(5) k(6) k(7) k(8)
                  k(2) k(1) k(8) k(7) k(6) k(5) k(4) k(3)
                  k(3) k(8) k(1) k(6) k(7) k(4) k(5) k(2)
                  k(4) k(7) k(6) k(1) k(8) k(3) k(2) k(5)
                  k(5) k(6) k(7) k(8) k(1) k(2) k(3) k(4)
                  k(6) k(5) k(4) k(3) k(2) k(1) k(8) k(7)
                  k(7) k(4) k(5) k(2) k(3) k(8) k(1) k(6)
                  k(8) k(3) k(2) k(5) k(4) k(7) k(6) k(1)];


function varargout = ndhcagui(varargin)
% NDHCAGUI M-file for ndhcagui.fig
%      NDHCAGUI, by itself, creates a new NDHCAGUI or raises the existing
%      singleton*.
%
%      H = NDHCAGUI returns the handle to a new NDHCAGUI or the handle to
%      the existing singleton*.
%
%      NDHCAGUI('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in NDHCAGUI.M with the given input arguments.
%
%      NDHCAGUI('Property','Value',...) creates a new NDHCAGUI or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before ndhcagui_OpeningFunction gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to ndhcagui_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help ndhcagui

% Last Modified by GUIDE v2.5 13-Apr-2006 00:52:45

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @ndhcagui_OpeningFcn, ...
                   'gui_OutputFcn',  @ndhcagui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT
```

```
% --- Executes just before ndhcagui is made visible.
function ndhcagui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to ndhcagui (see VARARGIN)

% Choose default command line output for ndhcagui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes ndhcagui wait for user response (see UIRESUME)
% uiwait(handles.ndhcafig);

global x c
disp('storing x...')
set(handles.resetndhca,'UserData',x);
disp('x stored')
if(~isempty(c)); cset = mean(mean(c)); else; cset=0; end
vtol = prod(size(x))*0.01*0.1;
mfset=mean(mean(x));
set(handles.cset,'String',cset);
set(handles.mfset,'String',mfset);
set(handles.vtol,'String',vtol);


% --- Outputs from this function are returned to the command line.
function varargout = ndhcagui_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% h andles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes during object creation, after setting all properties.
function cset_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function cset_Callback(hObject, eventdata, handles)
```

```
% --- Executes on button press in resetndhca.
function resetndhca_Callback(hObject, eventdata, handles)
global x
disp('retrieving x...')
x = get(hObject,'UserData');
disp('x retrieved')


% --- Executes on button press in runndhca.
function runndhca_Callback(hObject, eventdata, handles)
global x targets fixedU fixedF springK Ymodul Pratio pPower q                %*

csetcbx= get(handles.csetcbx,'Value');
if(csetcbx); cset = str2num(get(handles.cset,'String'))*targets; else cset=0;
end

bcond = get(handles.bcond,'Value');
scond= get(handles.scond,'Value');
mfsetcbx= get(handles.mfsetcbx,'Value');
if(mfsetcbx); mfset = str2num(get(handles.mfset,'String')); else mfset=0; end
backt = str2num(get(handles.backtedt,'String'));
itmax = str2num(get(handles.itmax,'String'));
vtol = str2num(get(handles.vtol,'String'));
% get convergence criteria type: local=0, global=1
if(get(handles.global_mass_cbx,'Value')==1); convergence_type=1; else;
convergence_type=0; end
movflg = get(handles.movcbx,'Value');
movstr = get(handles.movedt,'String');
switch get(handles.neigh,'Value');
    case 1; n = 0;
    case 2; n = 4;
    case 3; n = 8;
    case 4; n = 24;
    case 5; n = prod(size(x))-1;
end
k = [0 0 0 0 0 0 0];
if get(handles.kfcbx,'Value'); k(1) = str2num(get(handles.kfedt,'String'));end
if get(handles.kpcbx,'Value'); k(2) = str2num(get(handles.kpedt,'String')); end
if get(handles.kicbx,'Value');  k(3) = str2num(get(handles.kiedt,'String')); end
if get(handles.kdcbx,'Value');  k(4) = str2num(get(handles.kdedt,'String')); end
if get(handles.kecbx,'Value');  k(5) = str2num(get(handles.kiedt,'String')); end
if get(handles.kncbx,'Value'); k(6) = str2num(get(handles.knedt,'String')); end

mfsetcbx=get(handles.mfsetcbx,'Value');
if(mfsetcbx); gMf = str2num(get(handles.mfset,'String')); else gMf=0; end

mechanism=get(handles.meccbx,'Value');

tic
if ~mechanism
    x = ndhca(x, fixedF, fixedU, springK, Ymodul, Pratio, pPower, cset, gMf, k,
backt, n, bcond, scond, itmax, convergence_type, vtol, movflg, movstr, q);   %*
else
    x = ndhcam(x, fixedF, fixedU, springK, Ymodul, Pratio, pPower, cset, gMf, k,
backt, n, bcond, scond, itmax, convergence_type, vtol, movflg, movstr, q);
```

```
end
toc


% --- Executes on button press in closendhca.
function closendhca_Callback(hObject, eventdata, handles)
global x
disp('retrieving x...')
x = get(handles.resetndhca,'UserData');
disp('x retrieved')
close

% --- Executes during object creation, after setting all properties.
function ndhcafig_CreateFcn(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function kfedt_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function kfedt_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function itmax_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function itmax_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function vtol_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function vtol_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function neigh_CreateFcn(hObject, eventdata, handles)
if ispc
```

```
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


% --- Executes on selection change in neigh.
function neigh_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function kpedt_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function kpedt_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function backtedt_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function backtedt_Callback(hObject, eventdata, handles)


% --- Executes on button press in kfcbx.
function kfcbx_Callback(hObject, eventdata, handles)


% --- Executes on button press in kpcbx.
function kpcbx_Callback(hObject, eventdata, handles)


% --- Executes on button press in kdcbx.
function kdcbx_Callback(hObject, eventdata, handles)


% --- Executes on button press in checkbutton.
function checkbutton_Callback(hObject, eventdata, handles)
global x
disp('checking x...');
figure;
himgc = imagesc(-x,[-1 0]);
colormap(gray); axis equal; axis tight; axis off;
```

```
disp('x checked');


% --- Executes during object creation, after setting all properties.
function keedt_CreateFcn(hObject, eventdata, handles)
% hObject    handle to keedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function keedt_Callback(hObject, eventdata, handles)


% --- Executes on button press in kecbx.
function kecbx_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function kiedt_CreateFcn(hObject, eventdata, handles)
% hObject    handle to kiedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function kiedt_Callback(hObject, eventdata, handles)
% hObject    handle to kiedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of kiedt as text
%        str2double(get(hObject,'String')) returns contents of kiedt as a double


% --- Executes on button press in kicbx.
function kicbx_Callback(hObject, eventdata, handles)
% hObject    handle to kicbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of kicbx


% --- Executes during object creation, after setting all properties.
function kdedt_CreateFcn(hObject, eventdata, handles)
% hObject    handle to kdedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function kdedt_Callback(hObject, eventdata, handles)
% hObject    handle to kdedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of kdedt as text
%        str2double(get(hObject,'String')) returns contents of kdedt as a double


% --- Executes on button press in kdcbx.
function checkbox10_Callback(hObject, eventdata, handles)
% hObject    handle to kdcbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of kdcbx


% --- Executes during object creation, after setting all properties.
function knedt_CreateFcn(hObject, eventdata, handles)
% hObject    handle to knedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
```

```
function knedt_Callback(hObject, eventdata, handles)
% hObject    handle to knedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of knedt as text
%        str2double(get(hObject,'String')) returns contents of knedt as a double


% --- Executes on button press in kncbx.
function kncbx_Callback(hObject, eventdata, handles)
% hObject    handle to kncbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of kncbx


% --- Executes during object creation, after setting all properties.
function bcond_CreateFcn(hObject, eventdata, handles)
% hObject    handle to bcond (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


% --- Executes on selection change in bcond.
function bcond_Callback(hObject, eventdata, handles)
% hObject    handle to bcond (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = get(hObject,'String') returns bcond contents as cell array
%        contents{get(hObject,'Value')} returns selected item from bcond


% --- Executes during object creation, after setting all properties.
function movedt_CreateFcn(hObject, eventdata, handles)
% hObject    handle to movedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
```

```
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function movedt_Callback(hObject, eventdata, handles)
% hObject    handle to movedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of movedt as text
%        str2double(get(hObject,'String')) returns contents of movedt as a
double


% --- Executes on button press in movcbx.
function movcbx_Callback(hObject, eventdata, handles)
% hObject    handle to movcbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of movcbx


% --- Executes on button press in meccbx.
function meccbx_Callback(hObject, eventdata, handles)
% hObject    handle to meccbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of meccbx



function mfset_Callback(hObject, eventdata, handles)
% hObject    handle to mfset (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of mfset as text
%        str2double(get(hObject,'String')) returns contents of mfset as a double


% --- Executes during object creation, after setting all properties.
function mfset_CreateFcn(hObject, eventdata, handles)
% hObject    handle to mfset (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on button press in scond.
function scond_Callback(hObject, eventdata, handles)
% hObject    handle to scond (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of scond


% --- Executes on button press in mfsetcbx.
function mfsetcbx_Callback(hObject, eventdata, handles)
% hObject    handle to mfsetcbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of mfsetcbx

if(get(handles.csetcbx,'Value')==1)
    set(handles.csetcbx,'Value',0);
end

% --- Executes on button press in csetcbx.
function csetcbx_Callback(hObject, eventdata, handles)
% hObject    handle to csetcbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of csetcbx


if(get(handles.mfsetcbx,'Value')==1)
    set(handles.mfsetcbx,'Value',0);
end



% --- Executes on button press in global_mass_cbx.
function global_mass_cbx_Callback(hObject, eventdata, handles)
% hObject    handle to global_mass_cbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of global_mass_cbx

global x

if(get(handles.local_mass_cbx,'Value')==1)
```

```
    set(handles.local_mass_cbx,'Value',0);
else
    set(handles.global_mass_cbx,'Value',1);
end

vtol = prod(size(x))*0.01*0.1;
set(handles.vtol,'String',vtol);

% --- Executes on button press in local_mass_cbx.
function local_mass_cbx_Callback(hObject, eventdata, handles)
% hObject    handle to local_mass_cbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of local_mass_cbx

if(get(handles.global_mass_cbx,'Value')==1)
    set(handles.global_mass_cbx,'Value',0);
else
    set(handles.local_mass_cbx,'Value',1);
end

set(handles.vtol,'String',0.01);


function newcells = ndhcao(cells, fixedF, fixedU, E, nu, penal, cset, k,
neighbors, itmax, ctol)
% NDHCA0 Hybrid Cellular Automata Local Optimization
%
% Andres Tovar - Univ. of Notre Dame
% modified: Dec. 07, 2004 (03 h.)
% modified: Jan. 23, 2004 (12 h.)
% modified: Jan. 28, 2004 (21 h.) - initial parameters compatibel with ndfea
% modified: Jan. 29, 2004 (14 h.) - cleaning project old saved as ndhca129.m
% modified: Jan. 30, 2004 (14 h.) - k, itmax and ctol added
% modified: Feb. 24, 2004 (21 h.) - add tcontrol type of control FDC
% (2^2+2^1+2^0) fixed, proportional, cumulative

% INPUT PARAMETERS
[nely, nelx] = size(cells);
loadcases = max(fixedF(:,3));

% convergence criteria for change in density
vtol = nelx*nely*min(k)*ctol;

% IDENTIFICATION OF PASSIVE AND ACTIVE ELEMENTS
passive0 = (cells==0);
passive1 = (cells==1);
cells(find(passive0)) = 0.001;
cells(find(passive1)) = 1.000;

% START ALGORITHM
warning off MATLAB:nearlySingularMatrixUMFPACK
disp('running ndhcao...')
```

```
figure; colormap(gray);
title('running ndhcao...'); axis off; pause(1e-6);
% function call and intial strain energy c
[U, KE] = ndfea(cells, fixedF, fixedU, E, nu, penal);
c = sparse(nely,nelx);
it = 0;
for ely = 1:nely
    for elx = 1:nelx
        n1 = (nely+1)*(elx-1)+ely;
        n2 = (nely+1)* elx   +ely;
        dc(ely,elx) = 0.;
        for lcase=1:loadcases
            Ue = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],
lcase);
            c(ely,elx) = c(ely,elx) + cells(ely,elx)^penal*Ue'*KE*Ue;
            dc(ely,elx) = dc(ely,elx) - penal*cells(ely,elx)^(penal-
1)*Ue'*KE*Ue;
        end
    end
end
% plot volume fraction
volh = imagesc(-cells,[-1 0]); axis equal; axis tight; axis off;
str = [ ' t ' sprintf('%4i',it) ' c ' sprintf('%10.4f',full(sum(sum(c)))) ...
        ' V ' sprintf('%6.3f',sum(sum(cells)))];
title(str); pause(1e-6);
disp(str);

% ITERATIONS
cavgt = cset/(nelx*nely);
changev = inf;
while (it<itmax & changev>vtol)% & changec>ctol)
    it = it + 1;
    % SUMS AROUND EACH CELL -> csum (matrix)
    cavgn = doavg(c,neighbors);
    % CA OC RULE -> newcells
    newcells = max(0.001,max(cells-0.2,min(1.,min(cells+0.2,cells.*sqrt(-
dc./1)))));
    %check density limits
    newcells0 = (newcells < 0.001);
    newcells1 = (newcells > 0.999);
    newcells(find(newcells0)) = 0.001;
    newcells(find(newcells1)) = 0.999;
    % force density value in passive cells
    newcells(find(passive0)) = 0.001;
    newcells(find(passive1)) = 1.000;
    % function call - strain energy newc
    U = ndfea(newcells, fixedF, fixedU, E, nu, penal);
    newc = sparse(nely,nelx);
    for ely = 1:nely
        for elx = 1:nelx
            n1 = (nely+1)*(elx-1)+ely;
            n2 = (nely+1)* elx   +ely;
            newdc(ely,elx) = 0.;
            for lcase=1:loadcases
                Ue = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2+2; 2*n1+1;2*n1+2],
lcase);
```

```
                newc(ely,elx) = newc(ely,elx) +
newcells(ely,elx)^penal*Ue'*KE*Ue;
                newdc(ely,elx) = newdc(ely,elx) -
penal*newcells(ely,elx)^(penal-1)*Ue'*KE*Ue;
            end
        end
    end
    % convergence criteria
    changev = sum(sum(abs(newcells-cells)));
    %changec = sum(sum(abs(newc-c)));
    % update
    cells = newcells;
    c = newc;
    dc = newdc;
    % plot
    set(volh,'cdata',-cells)
    str = [ ' t ' sprintf('%4i',it) ' c ' sprintf('%10.4f',full(sum(sum(c))))
...
            ' V ' sprintf('%6.3f',sum(sum(cells)))];
    title(str); pause(1e-6);
    disp(str);
end
cells(find(passive0)) = 0;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function cavgn = doavg(c,neighbors)
[nely,nelx] = size(c);
csum = c; % no Neighborhood (N=0)
if neighbors > 0 % von Newmann (N=4)
    % extends c field for fixed BC
    cext(nely+2, nelx+2) = 0;
    cext(2:nely+1, 2:nelx+1) = c;
    y = 2:nely+1; x = 2:nelx+1;
    csum = csum + cext(y,x+1) + cext(y,x-1) + cext(y+1,x) + cext(y-1,x);
    if neighbors > 4 % Moore (N=8)
        csum = csum + cext(y+1,x+1) + cext(y-1,x-1) + cext(y+1,x-1) + cext(y-
1,x+1);
        if neighbors > 8 % Extended Moore (N=24)
            cextm(nely+4, nelx+4) = 0;
            cextm(3:nely+2, 3:nelx+2) = c;
            y = 3:nely+2; x = 3:nelx+2;
            csum = csum + ...
                cextm(y-2, x-2) + cextm(y-2, x-1) + cextm(y-2, x) + cextm(y-2,
x+1) + cextm(y-2, x+2) +...
                cextm(y+2, x-2) + cextm(y+2, x-1) + cextm(y+2, x) + cextm(y+2,
x+1) + cextm(y+2, x+2) +...
                cextm(y-1, x-2) + cextm(y, x-2) + cextm(y+1, x-2) + ...
                cextm(y-1, x+2) + cextm(y, x+2) + cextm(y+1, x+2);
            if neighbors > 24 % Global (nelx*ney-1)
                csum = csum*0 + ones(nely,nelx)*sum(sum(c));
            end

        end
    end
end
```

```
cavgn = csum./(neighbors+1); % average csum (matrix)



function varargout = ndtopgui(varargin)
% NDTOPGUI M-file for ndtopgui.fig
%      NDTOPGUI, by itself, creates a new NDTOPGUI or raises the existing
%      singleton*.
%
%      H = NDTOPGUI returns the handle to a new NDTOPGUI or the handle to
%      the existing singleton*.
%
%      NDTOPGUI('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in NDTOPGUI.M with the given input arguments.
%
%      NDTOPGUI('Property','Value',...) creates a new NDTOPGUI or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before ndtopgui_OpeningFunction gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to ndtopgui_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to runndtop (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help ndtopgui

% Last Modified by GUIDE v2.5 24-Apr-2004 19:10:29

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @ndtopgui_OpeningFcn, ...
                   'gui_OutputFcn',  @ndtopgui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT



% --- Executes just before ndtopgui is made visible.
function ndtopgui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to ndtopgui (see VARARGIN)

% Choose default command line output for ndtopgui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes ndtopgui wait for user response (see UIRESUME)
% uiwait(handles.ndtopguifig);

global x
disp('storing x...')
set(handles.resetndtop,'UserData',x);
disp('x stored')
set(handles.volfrac,'String',mean(mean(x)));

% --- Outputs from this function are returned to the command line.
function varargout = ndtopgui_OutputFcn(hObject, eventdata, handles)
varargout{1} = handles.output;


% --- Executes during object creation, after setting all properties.
function volfrac_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


% --- Executes during object creation, after setting all properties.
function penal_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function volfrac_Callback(hObject, eventdata, handles)

function rmin_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function rmin_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
```

```
% --- Executes on button press in resetndtop.
function resetndtop_Callback(hObject, eventdata, handles)
global x
disp('retrieving x...')
x = get(hObject,'UserData');
disp('x retrieved')

% --- Executes on button press in runndtop.
function runndtop_Callback(hObject, eventdata, handles)
global x fixedF fixedU Ymodul Pratio pPower
volfrac = str2num(get(handles.volfrac,'String'));
rmin = str2num(get(handles.rmin,'String'));
itmax = str2num(get(handles.itmax,'String'));
ctol = str2num(get(handles.ctol,'String'));
movflg = get(handles.movcbx,'Value');
movstr = get(handles.movedt,'String');
tic
if ~get(handles.meccbx,'Value')
    x = top99(x, fixedF, fixedU, Ymodul, Pratio, pPower, volfrac, rmin, itmax,
ctol, movflg, movstr);
    disp('top99 is done')
else
    x = top99m(x, fixedF, fixedU, Ymodul, Pratio, pPower, volfrac, rmin, itmax,
ctol, movflg, movstr);
    disp('top99m is done')
end
toc

% --- Executes on button press in closendtop.
function closendtop_Callback(hObject, eventdata, handles)
global x
disp('retrieving x...')
x = get(handles.resetndtop,'UserData');
disp('x retrieved')
close

function resetndtop_ButtonDownFcn(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function ndtopguifig_CreateFcn(hObject, eventdata, handles)


function runndtop_ButtonDownFcn(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function resetndtop_CreateFcn(hObject, eventdata, handles)


% --- Executes when user attempts to close ndtopguifig.
function ndtopguifig_CloseRequestFcn(hObject, eventdata, handles)
delete(hObject);
```

```
% --- Executes during object creation, after setting all properties.
function itmax_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function itmax_Callback(hObject, eventdata, handles)
% hObject    handle to itmax (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of itmax as text
%        str2double(get(hObject,'String')) returns contents of itmax as a double


% --- Executes during object creation, after setting all properties.
function ctol_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function ctol_Callback(hObject, eventdata, handles)

% --- Executes on button press in checkbutton.
function checkbutton_Callback(hObject, eventdata, handles)
global x
disp('checking x...');
figure;
himgc = imagesc(-x,[-1 0]);
colormap(gray); axis equal; axis tight; axis off;
disp('x checked');


% --- Executes during object creation, after setting all properties.
function movedt_CreateFcn(hObject, eventdata, handles)
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function movedt_Callback(hObject, eventdata, handles)
% hObject    handle to movedt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of movedt as text
```

```
%        str2double(get(hObject,'String')) returns contents of movedt as a
double


% --- Executes on button press in movcbx.
function movcbx_Callback(hObject, eventdata, handles)
% hObject    handle to movcbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of movcbx


% --- Executes on button press in meccbx.
function meccbx_Callback(hObject, eventdata, handles)
% hObject    handle to meccbx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of meccbx




function handles = plot_arrow( x1,y1,x2,y2,varargin )
%
% plot_arrow - plots an arrow to the current plot
%
% format:   handles = plot_arrow( x1,y1,x2,y2 [,options...] )
%
% input:    x1,y1   - starting point
%           x2,y2   - end point
%           options - come as pairs of "property","value" as defined for "line"
and "patch"
%                     controls, see matlab help for listing of these properties.
%                     note that not all properties where added, one might add
them at the end of this file.
%
%                     additional options are:
%                     'headwidth':  relative to complete arrow size, default
value is 0.07
%                     'headheight': relative to complete arrow size, default
value is 0.15
%                     (encoded are maximal values if pixels, for the case that
the arrow is very long)
%
% output:   handles - handles of the graphical elements building the arrow
%
% Example:  plot_arrow( -1,-1,15,12,'linewidth',2,'color',[0.5 0.5
0.5],'facecolor',[0.5 0.5 0.5] );
%           plot_arrow( 0,0,5,4,'linewidth',2,'headwidth',0.25,'headheight',0.33
);
%           plot_arrow;   % will launch demo
```

```
% ==========================================
% for debug - demo - can be erased
% ==========================================
% if (nargin==0)
%     figure;
%     axis;
%     set( gca,'nextplot','add' );
%     for x = 0:0.3:2*pi
%         color = [rand rand rand];
%         h = plot_arrow( 1,1,50*rand*cos(x),50*rand*sin(x),...
%             'color',color,'facecolor',color,'edgecolor',color );
%         set( h,'linewidth',2 );
%     end
%     hold off;
%     return
% end
% ==========================================
% end of for debug
% ==========================================


% ==========================================
% constants (can be edited)
% ==========================================
alpha       = 0.15;   % head length
beta        = 0.07;   % head width
max_length  = 22;
max_width   = 10;

% ==========================================
% check if head properties are given
% ==========================================
% if ratio is always fixed, this section can be removed!
if ~isempty( varargin )
    for c = 1:floor(length(varargin)/2)
        try
            switch lower(varargin{c*2-1})
                % head properties - do nothing, since handled above already
            case 'headheight',alpha = max( min( varargin{c*2},1 ),0.01 );
            case 'headwidth', beta = max( min( varargin{c*2},1 ),0.01 );
            end
        catch
            fprintf( 'unrecognized property or value for: %s\n',varargin{c*2-1}
);
        end
    end
end

% ==========================================
% calculate the arrow head coordinates
% ==========================================
den         = x2 - x1 + eps;                                    % make sure no
devision by zero occurs
teta        = atan( (y2-y1)/den ) + pi*(x2<x1) - pi/2;      % angle of arrow
cs          = cos(teta);                                        % rotation matrix
```

```
ss          = sin(teta);
R           = [cs -ss;ss cs];
line_length = sqrt( (y2-y1)^2 + (x2-x1)^2 );               % sizes
head_length = min( line_length*alpha,max_length );
head_width  = min( line_length*beta,max_length );
x0          = x2*cs + y2*ss;                               % build head
coordinats
y0          = -x2*ss + y2*cs;
coords      = R*[x0 x0+head_width/2 x0-head_width/2; y0 y0-head_length y0-
head_length];

% =========================================
% plot arrow  (= line + patch of a triangle)
% =========================================
h1          = plot( [x1,x2],[y1,y2],'k' );
h2          = patch( coords(1,:),coords(2,:),[0 0 0] );

% =========================================
% return handles
% =========================================
handles = [h1 h2];

% =========================================
% check if styling is required
% =========================================
% if no styling, this section can be removed!
if ~isempty( varargin )
    for c = 1:floor(length(varargin)/2)
        try
            switch lower(varargin{c*2-1})

             % only patch properties
            case 'edgecolor',   set( h2,'EdgeColor',varargin{c*2} );
            case 'facecolor',   set( h2,'FaceColor',varargin{c*2} );
            case 'facelighting',set( h2,'FaceLighting',varargin{c*2} );
            case 'edgelighting',set( h2,'EdgeLighting',varargin{c*2} );

            % only line properties
            case 'color'    , set( h1,'Color',varargin{c*2} );

            % shared properties
            case 'linestyle', set( handles,'LineStyle',varargin{c*2} );
            case 'linewidth', set( handles,'LineWidth',varargin{c*2} );
            case 'parent',    set( handles,'parent',varargin{c*2} );

            % head properties - do nothing, since handled above already
            case 'headwidth',;
            case 'headheight',;

            end
        catch
            fprintf( 'unrecognized property or value for: %s\n',varargin{c*2-1}
);
        end
    end
```

```
end



function htri = plot_spring (x, y, ttype, mycolor)
% htri = triangle (x, y, ttype, b)
%   x, y are the coordinates of the node
%   ttpe=1 is horizontal direction
%   ttpe=2 is vertical direction
%   b is a blue tone color b=1,2,...

h=0.2;
%if mycolor==[] mycolor=rand(1,3); end
% r = (1 - 1/b)/1.5;
% g = (1 - 1/b)/1.0;
% b = (1 + 1/b)/3.0;
if ttype == 1 %Horizontal
    x1 = x-h;
    x2 = x1-(h*cos(pi/6));
    x3 = x2-(h*cos(pi/6));
    x4 = x3-(h*cos(pi/6));
    x5 = x4-h/1.5;
    y1 = y;
    y2 = y1+h;
    y3 = y2-2*h;
    y4 = y3+h;
    y5 = y4;
    htri(1) = line([x,x1],[y,y],'Color',mycolor,'LineWidth',1.5);
    htri(2) = line([x1,x2],[y1,y2],'Color',mycolor,'LineWidth',1.5);
    htri(3) = line([x2,x3],[y2,y3],'Color',mycolor,'LineWidth',1.5);
    htri(4) = line([x3,x4],[y3,y4],'Color',mycolor,'LineWidth',1.5);
    htri(5) = line([x4,x5],[y4,y5],'Color',mycolor,'LineWidth',1.5);
    htri(6) = line([x5,x5],[y5-3*h/2,y5+3*h/2],'Color',mycolor,'LineWidth',1.5);
else if ttype == 2 %Horizontal
        x1 = x+h;
        x2 = x1+(h*cos(pi/6));
        x3 = x2+(h*cos(pi/6));
        x4 = x3+(h*cos(pi/6));
        x5 = x4+h/1.5;
        y1 = y;
        y2 = y1+h;
        y3 = y2-2*h;
        y4 = y3+h;
        y5 = y4;
        htri(1) = line([x,x1],[y,y],'Color',mycolor,'LineWidth',1.5);
        htri(2) = line([x1,x2],[y1,y2],'Color',mycolor,'LineWidth',1.5);
        htri(3) = line([x2,x3],[y2,y3],'Color',mycolor,'LineWidth',1.5);
        htri(4) = line([x3,x4],[y3,y4],'Color',mycolor,'LineWidth',1.5);
        htri(5) = line([x4,x5],[y4,y5],'Color',mycolor,'LineWidth',1.5);
        htri(6) = line([x5,x5],[y5-
3*h/2,y5+3*h/2],'Color',mycolor,'LineWidth',1.5);

    elseif ttype == 3 %Vertical
        y1 = y+h;
        y2 = y1+(h*cos(pi/6));
        y3 = y2+(h*cos(pi/6));
```

```
        y4 = y3+(h*cos(pi/6));
        y5 = y4+h/1.5;
        x1 = x;
        x2 = x1+h;
        x3 = x2-2*h;
        x4 = x3+h;
        x5 = x4;
        htri(1) = line([x,x],[y,y1],'Color',mycolor,'LineWidth',1.5);
        htri(2) = line([x1,x2],[y1,y2],'Color',mycolor,'LineWidth',1.5);
        htri(3) = line([x2,x3],[y2,y3],'Color',mycolor,'LineWidth',1.5);
        htri(4) = line([x3,x4],[y3,y4],'Color',mycolor,'LineWidth',1.5);
        htri(5) = line([x4,x5],[y4,y5],'Color',mycolor,'LineWidth',1.5);
        htri(6) = line([x5-
3*h/2,x5+3*h/2],[y5,y5],'Color',mycolor,'LineWidth',1.5);
    elseif ttype == 4 % Vertical
        y1 = y-h;
        y2 = y1-(h*cos(pi/6));
        y3 = y2-(h*cos(pi/6));
        y4 = y3-(h*cos(pi/6));
        y5 = y4-h/1.5;
        x1 = x;
        x2 = x1+h;
        x3 = x2-2*h;
        x4 = x3+h;
        x5 = x4;
        htri(1) = line([x,x],[y,y1],'Color',mycolor,'LineWidth',1.5);
        htri(2) = line([x1,x2],[y1,y2],'Color',mycolor,'LineWidth',1.5);
        htri(3) = line([x2,x3],[y2,y3],'Color',mycolor,'LineWidth',1.5);
        htri(4) = line([x3,x4],[y3,y4],'Color',mycolor,'LineWidth',1.5);
        htri(5) = line([x4,x5],[y4,y5],'Color',mycolor,'LineWidth',1.5);
        htri(6) = line([x5-
3*h/2,x5+3*h/2],[y5,y5],'Color',mycolor,'LineWidth',1.5);
    end
end


function htri = triangle (x, y, ttype, tcolor)
% htri = triangle (x, y, ttype, b)
%   x, y are the coordinates of the node
%   ttpe=1 is horizontal direction
%   ttpe=2 is vertical direction
%   b is a blue tone color b=1,2,...

h=1.0;
if ttype == 1 %Horizontal
    x1 = x-(h*cos(pi/6));
    y1 = y+(h/2);
    y2 = y-(h/2);
    htri(1) = line([x,x1],[y,y1],'Color', tcolor);
    htri(2) = line([x1,x1],[y1,y2],'Color', tcolor);
    htri(3) = line([x1,x],[y2,y],'Color', tcolor);
elseif ttype == 2 %Vertical
    x1 = x+(h/2);
    x2 = x-(h/2);
    y1 = y+(h*cos(pi/6));
    htri(1) = line([x,x1],[y,y1],'Color', tcolor);
```

```
    htri(2) = line([x1,x2],[y1,y1],'Color', tcolor);
    htri(3) = line([x2,x],[y1,y],'Color', tcolor);
end
```