Topology Optimization Applications for FSAE Wing Mounting Rib

Team 2: Aaron Burns, Annette Guo, Daniel Emerson, Kevin Dai

December 11th, 2020

Abstract

In the popular student racing competition Formula SAE, the implementation of inverted airfoils as part of the aerodynamics package provides highly desirable benefits such as increased traction and cornering abilities, as well as decreased drag. Such vehicles typically balance the goals of minimizing mass while maximizing performance, and can benefit from the use of topology optimization in the design of the airfoil rib structures to reveal where material can be removed. We utilize a Solid Isotropic Material with Penalization (SIMP) framework and `fmincon` nonlinear programming (NLP) solver to minimize compliance subject to stress and volume constraints and different race car loading conditions. Our results show the presence of many design solutions indicating many local minima and a nonconvex problem. We ultimately selected a design based on quantitative metrics such as volume fraction and factor of safety (FOS) and qualitative engineering assessment of intuitive design principles that informs where material is most required for future studies.

1 Introduction

In FSAE racing, an aerodynamics package can provide significant advantages over competition through elements such as inverted airfoils, which produce increased traction between the road and tires, better cornering abilities, and decreased drag. These airfoils typically contain aluminum mounting ribs, which are epoxied to the carbon fiber airfoils in order to constrain them to the desired shape and provide structural support for a given system of airfoils, which are collectively referenced to as a “wing”.

To maximize the benefits of the aerodynamic system, these aluminum mounting ribs require a high stiffness to mass ratio. This is because the ribs need to balance between adding minimal weight to the car and having an appropriate level of stiffness to withstand the forces and moments generated from the air pressure, as well as the weight of the wing itself. In this project, we apply topology optimization to the problem and its constraints to achieve this high stiffness to mass ratio in two stages: first, by minimizing the compliance of the system with the objective function (see Section 2), and second, by minimizing the mass by changing the volume fraction in a parametric study.

To solve this problem, we had to make decisions about our approach and the many different variables that can influence the solution. We decided to use the Solid Isotropic Material with Penalization (SIMP) framework since it is a common approach for topology optimization [2][1]. To solve the optimization problem, we chose to use `fmincon` since we had the most familiarity with the function, and chose to use the sequential quadratic programming (SQP) algorithm after comparing it with the interior-point algorithm. Within this framework, we decided to capture the airfoil shape with a quadrilateral mesh in order to account for the curvature and irregularity of the shape. Further decisions were made about where the loading conditions and active constraints where material was required. Lastly, we selected the variables volume fraction v_{f}, penalty value p, and sensitivity filter radius $r_{\min}$ for tuning in our sensitivity analysis.

This problem has a nontrivial solution because of its nature as a Non-Linear Programming (NLP) problem, and is non-convex due to the presence of different designs and local minima associated with different initial conditions. In addition, there is a lot of complexity to be found by adding in different parameterizations of the original shape of the airfoil such as mesh structure, mounting location, spar locations which allow for other airfoils to attach to the mounting rib. After a result is produced, this problem further requires some interpretation of the regions shown to have a partial volume fraction, which creates further challenges.

Finally, we can identify some key trade offs in the development of this problem, with the most obvious one being the aforementioned trade off between stiffness and mass. There are also considerable compromises to be made between model accuracy and computation time, in which we use an approximation of the gradient to reduce algorithm run-time. Another trade off is the 2D plane assumption, which can't account for forces in and out of the plane, but simplifies the model for our applications.

2 Problem Statement

Minimize the compliance of a finite element mesh with partial element densities when subjected to a set of prescribed loading conditions, stress constraints, and given volume fraction. We then conduct several parametric studies with the following parameters: volume fraction v_f, filtering radius r_min, and penalty factor p.

2.1 Negative Null Form

$$\begin{aligned}
&\text{minimize}\, f(\mathbf{x}) = c_1(\mathbf{x}) + c_2(\mathbf{x}) + c_3(\mathbf{x}) && \text{(overall compliance function)}\\
&\text{with respect to}\\
&\qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} && \text{(element density)}\\
&\text{subject to}\\
&\qquad \frac{\sum_{i=1}^n x_i a_i}{\sum_{i=1}^n a_i} - v_\mathrm{f} = 0 && \text{(volume fraction)}\\
&\qquad \boldsymbol{\sigma}(\mathbf{x})^{\mathrm{vM}} - \sigma_\mathrm{y} \leq 0 && \text{(von Mises stress constraint)}\\
&\qquad \frac{\sum_{i=1}^n x_i a_i^\mathrm{A}}{\sum_{i=1}^n a_i^\mathrm{A}} - 1 = 0 && \text{(active elements constraint)}\\
&\qquad -x_i \leq 0.001 && \text{(decision variable lower bound)}\\
&\qquad x_i \leq 1 && \text{(decision variable upper bound)}\\
&\qquad \mathbf{x} \in \mathbb{R}^n && \text{(decision variable is real valued)}\\
&\text{where}\\
&\qquad c_j(\mathbf{x}) = \sum_{i=1}^n (x_i)^p \mathbf{u}_i(\mathbf{x})^T \mathbf{K}_i \mathbf{u}_i(\mathbf{x}) && \text{(individual compliance function)}\\
&\qquad \mathbf{u}_j(\mathbf{x}) = \mathbf{K}(\mathbf{x})^{-1}\mathbf{f}_j && \text{(force-displacement equation)}\\
&\qquad \mathbf{K}(\mathbf{x}) = \sum_{i=1}^n (x_i)^p \mathbf{K}_i && \text{(stiffness matrix formulation)}\\
&\qquad \sigma^{\mathrm{vM}}(x_i) = {x_i}^{\frac{1}{2}} \hat{\sigma}_i^{\mathrm{vM}} && \text{(penalized von Mises stress)}\\
&\qquad \hat{\sigma}_i^{\mathrm{vM}} = \sqrt{\hat{\sigma}_{1,i}^2 - \hat{\sigma}_{1,i}\hat{\sigma}_{2,i} + \hat{\sigma}_{2,i}^2 + 3\hat{\tau}_{12,i}^2} && \text{(von Mises stress)}\\
&\qquad \hat{\sigma}_i = \mathbf{E}\varepsilon_i && \text{(Hooke's Law)}\\
&\qquad \varepsilon_i = \mathbf{B}\mathbf{u}(x_i) && \text{(strain-displacement)}
\end{aligned}$$

2.2 SIMP Framework

We used a modified form of Solid Isotropic Material with Penalization (SIMP) method for topology optimization as described in [7]. The SIMP method first discretizes the design domain into a finite number of n

Table 1: Model Parameters

Symbol	Description	Value	Units
n	number of mesh elements	1073	-
$\mathbf{a}$	mesh element area	-	m^2
$\mathbf{f}_j$	individual load case force vector	-	N
$\mathbf{x}^{\mathrm{A}}$	active element indicator vector	-	-
p	penalty parameter	[1.5, 3, 4.5]	-
v_{f}	volume fraction	[0.4, 0.6, 0.8]	-
$r_{\min}$	filtering radius	[0.003, 0.006, 0.009]	m
σ_{y}	yield stress	2.76×10^8	N/m^2
E	Young's modulus	6.89×10^{10}	N/m^2
ν	Poisson's ratio	0.33	-

elements. We decided to discretized our domain with a mesh of non-uniform quadrilateral elements. Each element has a density $x_i \in [0.001, 1]$ corresponding to the presence of material. With every iteration of the optimizer, values of $\mathbf{x}$ change until the solution converges. In an ideal world $\mathbf{x} = 0$ or $\mathbf{x} = 1$, indicating material or no material. However this would be a MINLP and would be difficult to solve. Therefore, we simplify the problem by assuming a continuous range of values for $\mathbf{x}$.

$$\mathbf{K}_e = (x_i)^p \mathbf{K}_i \tag{1}$$

Eqn. 1 shows how material properties vary with respect to $\mathbf{x}$, where $\mathbf{K}_i$ is the default stiffness matrix for a given quadrilateral element. $\mathbf{K}_i$ varies with each individual quadrilateral element's shape. The individual element stiffness matrix $\mathbf{K}_i$ is scaled according to the elemental density x_i raised to the power of the penalization factor p to obtain $\mathbf{K}_e$, the scaled individual element stiffness matrix. The penalization factor is typically set to 3, and as a result drives the value of $(x_i)^p$ closer to a value of 0 or 1. We will explore various values for p later in this paper. This penalty helps push the solution of the problem to be more binary even though we chose to avoid the true MINLP formulation.

The values of $\mathbf{u}(\mathbf{x})$ and $\mathbf{K}(\mathbf{x})$ change with every iteration of the design, and are generated by a call to a finite element method (FEM) subroutine. Specifically, we will use the direct stiffness method which is described in Eqn. 2 and explained thoroughly in [6].

$$\mathbf{u}_j(\mathbf{x}) = \mathbf{K}(\mathbf{x})^{-1}\mathbf{f}_j \tag{2}$$

Given that we know the applied forces $\mathbf{f}_j$ and stiffness of the structure $\mathbf{K}$, we aim to solve for the displacement vector $\mathbf{u}_j$ with every call to the FEM subroutine. This subroutine accounts for fixed model parameters such as Young's modulus E, Poisson's ratio ν, individual loading cases $\mathbf{f}_j$, and fixed displacement nodes (supports). The material is assumed to be Aluminum 6061 with $\sigma_y = 2.76 \times 10^8 N/m^2$, $E = 6.89 \times 10^{10} N/m^2$, and $\nu = 0.33$.

The value of the objective function $f(\mathbf{x})$, the compliance or stiffness, of the structure can be computed with the new values of $\mathbf{u}(\mathbf{x})$ and $\mathbf{K}(\mathbf{x})$. The sensitivity function in Eqn. 3 is used as a replacement gradient to substitute the gradient of the compliance objective function. Comparison with the objective function (see Eqn. 6) shows that the sensitivity function does not treat $\mathbf{u}_i$ as a function of $\mathbf{x}$. A true gradient would require chain rule to calculate $\partial \mathbf{u}_i(\mathbf{x})/\partial \mathbf{x}$ but is very difficult to calculate due to the size of the associated matrices. Our attempt to calculate the true gradient using MATLAB's Symbolic Toolbox took over 40 hours of computational time without completion. Additionally, running the problem with finite differences sees an extreme increase in computation time from $1-5$ minutes per solution with the approximate gradient to 5 hours per solution in the finite differences case.

$$\frac{\partial f}{\partial x_i} = -p(x_i)^{p-1}\mathbf{u}_i^T \mathbf{K}_i \mathbf{u}_i \tag{3}$$

To better ensure the existence of solutions, a filtering technique is used when building the new density matrix $\mathbf{x}$ as described in [7]. This filter shown in Eqn. 4 modifies the sensitivities from Eqn. 3 to take into account the densities of neighboring elements. This has the effect of reducing the checkerboard effect that can appear in solutions and helps to define the minimum feature size. Trusses that appeared in the optimization solution tended to have a width similar to r_{min}.

$$\widehat{\frac{\partial f}{\partial x_i}} = \frac{\sum_{k=1}^{n} \hat{h}_k x_k \frac{\partial f}{\partial x_k} / a_k}{x_i / a_i \sum\limits_{k=1}^{n} \hat{h}_k} \tag{4}$$

where $\hat{h}_k = r_{min} - \text{dist}(i,k)$

and a_i is the area of an element

The distance between elements $\text{dist}(i,k)$ is measured between the centroids of elements i and k. Elements are only considered if their distance $\text{dist}(i,k)$ is smaller than the filter radius r_{min}. Basically, the sensitivity of each mesh element becomes a weighted average of its neighboring elements' sensitivities, with the nearest elements having the heaviest influence. Elements are also weighted based on their area.

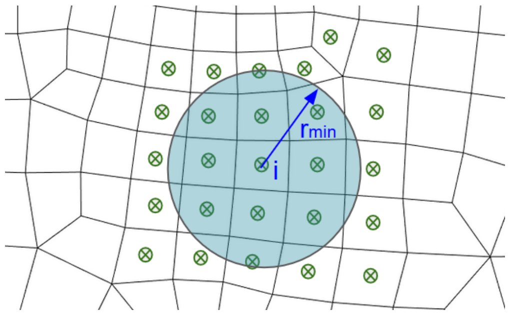


Figure 1: Circle showing the sensitivity filter radius r_{min} and centered at the centroid of an element i. Elements whose centroid fall within the circle will be used as elements k in Eqn. 4.

2.3 Stress Constraint

The SIMP method detailed in Section 2.2 is stiffness-based and does not include constraints or calculations for stress in the design. It is possible for the design solution to have a peak stress that exceeds the yield stress of the material. To prevent this undesirable condition, we implemented a linear inequality stress constraint for each element in the `fmincon` optimization algorithm. The stress constraints introduce a stress penalization function in addition to the stiffness penalization function in Eqn. 1, which will help force x_i to 0 or 1 [4].

$$\sigma_i = {x_i}^{\frac{1}{2}} \hat{\sigma}_i^{\text{vM}} \tag{5}$$

where $\hat{\sigma}_i^{vM}$ is the von Mises stress for an element calculated from FEA and σ_i is the penalized stress.

2.4 Airfoil Parameterization

For our first iteration with topology optimization for the airfoil, we used a grid of uniform square elements. In order to properly apply topology optimization to this grid, we defined the 2-dimensional profile of the airfoil rib within the grid, also known as Airfoil Parameterization. The objective was to define a boundary where decision variables are either included or excluded from the optimization algorithm by incorporating the "passive elements" section of the 99-line code optimization algorithm [7]. The passive elements effectively remove areas from consideration by assigning them a density of 0.001. In this project, the airfoil shape is fixed with defined coordinates that were processed through MATLAB's polyfit function to create representative polynomials. In the interior of the airfoil there are also two spar locations where material cannot be allocated and therefore must be excluded from the optimization algorithm. These locations are 1" and 0.75" diameter circular sections and were defined by their location in the 2-dimensional airfoil profile using standard circular equations.

2.5 Quadrilateral Finite Elements

Although the SIMP method traditionally utilizes a mesh grid with uniform square elements, our problem has a defined outer profile for the airfoil geometry. We decided to create a dedicated mesh using ANSYS with the goal of achieving more detailed solutions for our second iteration of topology optimization. This also has an added benefit of increasing computational efficiency because a dedicated mesh does not consume computational resources to define passive elements of the the airfoil profile.

The 2D mesh was generated in ANSYS Workbench 2020 R2 with a Uniform Surface Mesh Method and preference for All Quadrilateral elements. We also included an edge sizing of 5mm on the outer profile and targeted an element size of 5mm. Despite the All Quadrilateral element setting, some triangular elements were created in the mesh but this did not have a significant impact on implementation. Two lists of elements with their respective nodes, and nodes with their respective positions, were exported from ANSYS and imported into MATLAB (Fig. 2). This allowed us to perform topology optimization within MATLAB. We use a MATLAB function `polygeom.m` [8] to calculate the centroid and area of the quadrilateral mesh elements. We also use a MATLAB function `meshPlot.m` [3] to generate 2D plots of the quadrilateral mesh (e.g. Fig 2).

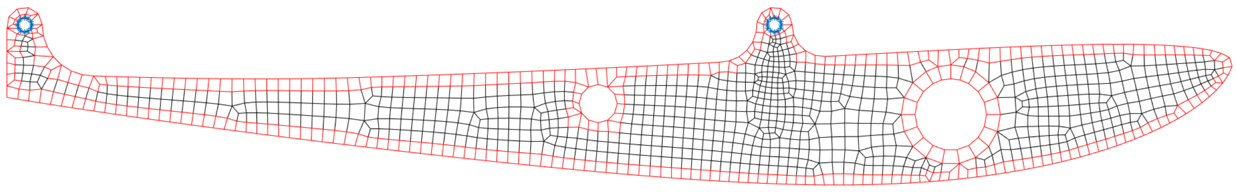


Figure 2: Quadrilateral element mesh, generated in ANSYS and imported to MATLAB. Elements outlined in red are active and will be set to have a density of 1. Blue asterisks mark the nodes with fixed displacement.

We then implemented calculations in MATLAB for linear quadrilateral finite elements (QUAD4). The isoparametric, 4-node quadrilateral finite elements required different element stiffness matrices, $\mathbf{K}_i$, than square elements. Calculation of the stiffness matrices requires several supporting equations that were taken from "Introduction to the Finite Element Method" by Nikishkov [5], including shape functions, a strain-displacement matrix $\mathbf{B}$, a Jacobian matrix, and Gauss quadrature to replace integration. The strain-displacement matrix was also used to calculate strain at the isoparametric center of each element, which in turn was used to calculate element stress. We assume plane stress for the stiffness and material properties.

Since we have a defined outer profile of the airfoil geometry, we knew that we wanted full element density for any elements that were touching the airfoil profile. We call these "Active elements" and highlight them red in Figure 2. Active elements are constrained to have $x_i = 1$. Elements around the spars are also set to be active.

2.6 Load Cases and Boundary Conditions

The CMU Racing vehicle is expected to accelerate, brake, and turn around corners during races. In each of these cases, the airfoil will experience different loading conditions on different aspects of the airfoil. For example, during acceleration the vehicle may initially be at a low velocity and not experience much down force or drag force. Or during braking, the vehicle may be at high velocity and experience a large amount of down force along with inertial loads. We chose to analyze three different load cases: one for acceleration, one for steady state velocity, and one for braking. We did not analyze cornering loads because they would require out of plane loads that could not be simulated by our 2D elements with our assumptions. The loads were applied as distributed forces at nodes on either the airfoil surface or the spar holes (Fig. 3). For topology optimization, we decided to concurrently simulate each of the load cases and equally weight their contributions to the objective function.

There are two small holes on the airfoil's tabs that are used to mount the airfoil rib to the vehicle. In the mesh, the nodes on these holes have a fixed displacement boundary conditions so that they cannot move. These nodes are marked with asterisks in Figure 3.

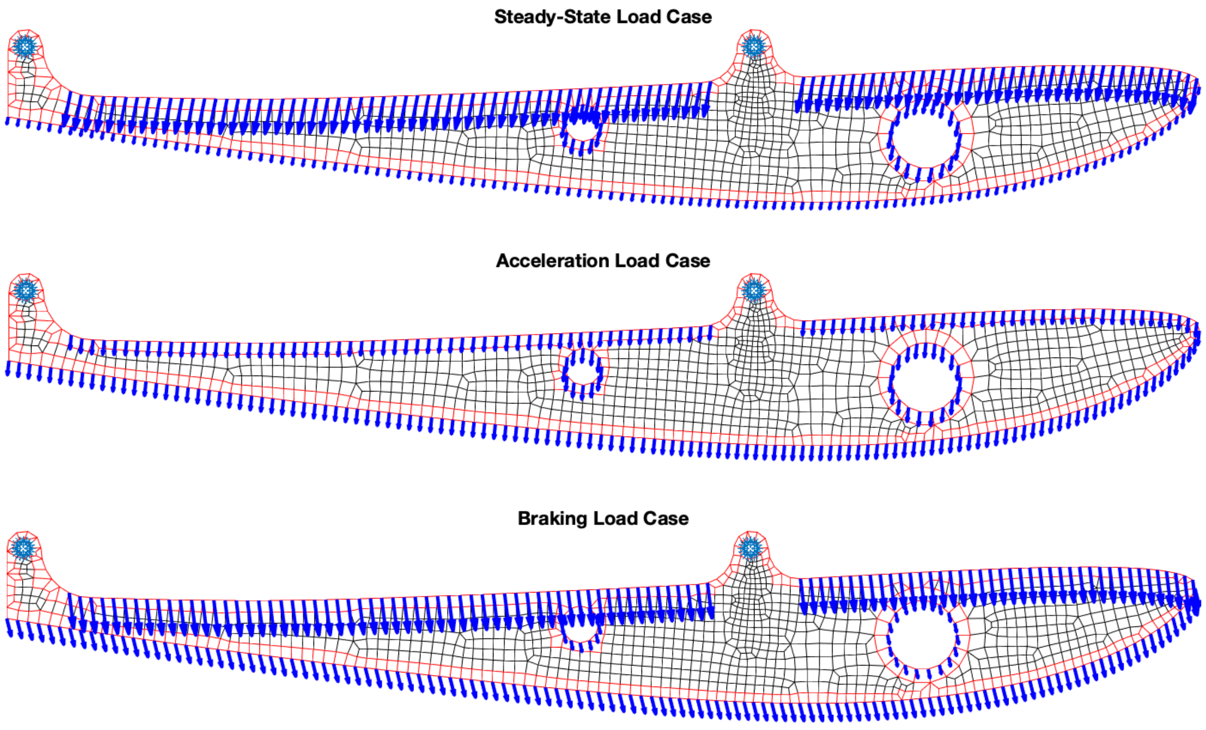


Figure 3: Nodal forces for each of the three load cases: steady state velocity, acceleration, and braking.

3 Analysis of Problem Statement

3.1 Model Assumptions

Since our decision variable is element density, our model makes the assumption that $0.001 \leq x_i \leq 1$, most notably that no element can actually equal 0 ($x_i \neq 0$). This assumption is made to avoid singularity when inverting the global stiffness matrix $\mathbf{K}$ [7]. It also must be assumed that there is a minimum amount of elements that have a significant amount of material (value of 1) so that an appropriate solution can be found. To this end, we will define all elements on the border of the airfoil, and around mounting and spar location to have material throughout the entire solving process. This is a practical constraint to preserve the aerodynamics of the airfoil surface and allow for the airfoil to be properly mounted at mounting and spar locations.

In order to use the SIMP Method, we must also make the assumption that the rib is made out of an isotropic material, since the method requires constant material properties to "discretize the design domain" [7]. Lastly, we will also assume that we are optimizing the rib geometry in 2D in order to first simplify the problem with plane stress equations. No regions of the feasible domain exist where the modeling assumptions do not apply, as they apply over all possible elements.

3.2 Natural vs. Practical Constraints

The force-displacement, stress, and upper bounds on x_i are natural constraints whereas the volume fraction, active elements, and lower bound on x_i are practical constraints. The force-displacement and stress constraints are defined by physics equations that represent natural phenomenon without artificially imposed limits. In addition, the upper bound on x_i represents the logical maximum proportion of 1 for material at each point in the mesh grid where x_i values above 1 are either unintended or not able to be interpreted. Volume fraction is modeled as a practical constraint to support our methodology of assessing different airfoil designs while iterating through parameter values. In addition, our active elements are represented as practical constraints which intend to preserve element density ($x_i = 1$) on the border regions of the airfoil e.g. the outside border and the internal spar locations. Finally, we practically limit the lower bound on x_i to prevent possible singularity issues associated with the SIMP method [7].

3.3 Problem Classification

This topology optimization project is classified as a nonlinear programming problem due to the nonlinear physics equations which create the force, displacement, and stiffness matrices, the exponential penalty parameter, and the exponential feature of the stress constraint. Furthermore, as mentioned in Section 2.2, by allowing the mesh of decision variables x_i to be any value between $[0.001, 1]$ instead of strictly integer values, the problem is not classified as an Integer Nonlinear Program (INLP).

3.3.1 Continuity

Our problem formulation makes several efforts to avoid discontinuities. Firstly, we restrict $\mathbf{x} \geq 0.001$ to avoid issues caused by dividing by an element density of 0 when solving for $\mathbf{u}_i(\mathbf{x})$ in the FEM subroutine. Additionally we apply a maximum penalty parameter p of 4.5 in our parametric studies. At higher values the penalty parameter caused issues with singularities when inverting the force-displacement equation in the FEM subroutine (see Eqn. 2). Additional areas for concern regarding discontinuity might include the volume fraction and active elements constraints described in Section 2.1. In volume fraction constraint, the sum of the element areas $\sum_{i=1}^{n} a_i$ is a scalar value and not equal to zero. In the active elements constraint we have a similar situation. The denominator is just the sum of the active elements indicator vector, $\sum_{i=1}^{n} x_i^{\mathrm{A}}$. This sum is also scalar greater than zero, precisely it is the number of active elements we expect in the design.

3.3.2 Smoothness

We cannot guarantee the smoothness of the problem posed in Section 2.1. Namely the gradient of our objective function, Eqn. 6 is too complex to calculate. Specifically, the FEM calculation of $\mathbf{u}_j(\mathbf{x})$ is highly complex and difficult to take the gradient of. As mentioned previously in Sec 2.2, we attempted to compute this gradient using MATLAB's Symbolic Toolbox and obtained no result after 40 hours of computation time.

$$\nabla_{\mathbf{x}} f = \frac{\partial}{\partial \mathbf{x}} \left(\sum_{j=1}^{3} \sum_{i=1}^{n} (x_i)^p \mathbf{u}_{i,j}(x_i)^{\mathrm{T}} \mathbf{K}_i \mathbf{u}_{i,j}(x_i) \right) \tag{6}$$

Alternatively, we could allow MATLAB's `fmincon` to compute this gradient using the finite difference method. Again, this takes an extended amount of computation time; about 5 hours per simulation. Instead we elected to provide an approximation for the gradient as described in Eqn. 3. This approximate gradient is continuous, thus guaranteeing the smoothness of our adapted problem. There are some potential downsides associated with supplying the solver with an inaccurate gradient, such as leading the solver away from the correct search direction. Given the complexity of the problem, and the issues mentioned previously, we ultimately made the decision to stick with our replacement gradient.

3.3.3 Convexity

Our problem is not convex as evident by the presence of many unique design solutions. Depending on the input parameters; penalty factor p, filtering radius $r_{\min}$, and volume fraction v_f, the algorithm arrives at unique solutions to the problem, reference Figure 7. Even though the `fmincon` solutions only claim to *indicate* the possible presence of a local minima, given the widespread occurrence, we conclude that the problem has many local minima and is therefore not convex. To further support this notion, if we initialize the solver with $\mathbf{x}_0$ equal to the previously found solution and lower the limiting termination criteria, minimum search direction step length, the solver remains at the same solution, indicating some level of stability to the solutions.

3.3.4 Undefined Regions

Our problem is undefined without clear interpretation for $x \notin [0.001, 1]$. Additionally, due to the active elements constraint, the problem is not defined for volume fractions below 0.31, or $v_{\mathrm{f}} \not< 0.31$. This is because we constrain all active elements to have a density of $x_i = 1$, and the area of all active elements sums to 31% of the total mesh area. Any volume fraction below 0.31 would cause a conflict between the volume fraction constraint and active elements constraint.

3.3.5 Size

There are 1073 decision variables which is equivalent to the number of quadrilateral elements in the mesh. In addition, there are four key parameters including the penalty factor p, filtering radius $r_{\min}$, volume fraction v_{f}, and the optimization starting location x_0. The problem formulation contains a linear inequality von Mises stress constraint and upper and lower bound for each element (1073 total). There are also two additional nonlinear equality constraints which represent the required volume fraction and ensure active elements along design boundaries retain an element density of 1 during the optimization process.

3.4 Problem Reformulation

There are two examples of reformulation in this study which included adapting bounds on the decisions variables and using a sensitivity function as a replacement for the gradient of the objective function. The decision variables represent each individual element density and in order to manufacture a true airfoil structure the densities should be either 0 or 1. However, allowing elements to be zero caused issues in the executing the SQP algorithm, possibly due to singularity issues [7]. As a result, the lower bound on the decision variables was set to a small value of 0.001. The second reformulation concerns calculating the gradient of the objective function. `fmincon` experienced significant processing time increases when required to calculate the gradient as opposed to using a user provided equation. Efforts to manually calculate the predetermined gradient function were not successful and as a result, an objective sensitivity function, common to topology optimization with an optimality criteria optimization method, was applied instead [7]. The sensitivity function is shown in equation (3). Inclusion of the sensitivity function significantly reduced processing time and helped generate a reasonable solution.

3.5 Scaling Implementation

Scaling was not incorporated into this study primarily due to the range of values available to the decisions variables. Each variable has the same constrained range between (0.001, 1] removing disparities between decisions variables. A possible subject of follow-on study may be to investigate the impact of scaling the stress constraints. The stress calculations can be in excess of 200×10^6 N/m^2 and may produce different results if the scale is reduced.

4 Optimization Study

4.1 Numerical Method & Software

The numerical method Sequential Quadratic Programming (SQP) was used in conjunction with MATLAB's `fmincon` to solve the optimization problem. This seemed to be the easiest and most straightforward method to implement SQP for topology optimization, as a literature review showed more challenging methods to implement SQP that would not have fit our schedule. SQP was also selected over the interior point method since it ran faster and consistently produced a more manufacturable result. Since the problem was nonconvex, we implemented multiple parameter conditions to search for a global minimum, and were able to locate a a variety of local minima. Reference Section 5.2 and Figure 7 for a discussion of the designs and global optimality.

4.2 Base Case Solution

To choose the best design, we examined the Factor of Safety, the objective function value representing compliance, and applied engineering judgement based on manufacturability and the presence of intuitive design elements such as trusses. Our selected solution had a compliance value of 17.71 and is in Figure 4. Associated input parameters and output metrics are located in Table 4.2 While this may not have been a global minimum, this solution showed commonalities with other solutions and provided clear guidance on truss placements.

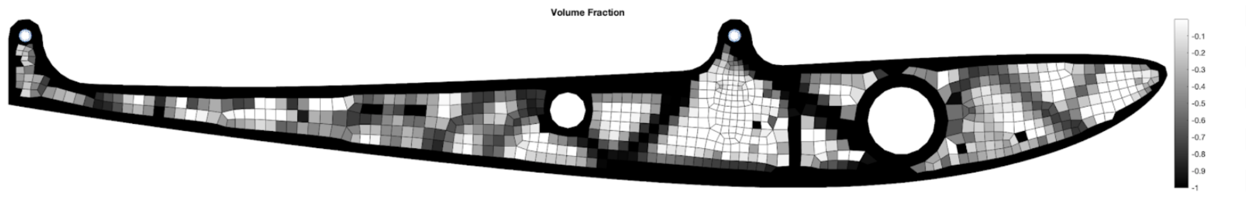


Figure 4: Selected base case solution with compliance of 17.71 and FOS of 3.15.

Table 2: Input and output parameters of chosen design

Input Parameters		Output Parameters	
penalty	4.5	`fval`	17.71
$r_{\min}$	$0.006\ m$	Max von Mises Stress	$87.5\ MPa$
v_{f}	0.6	`fmincon` iterations	4

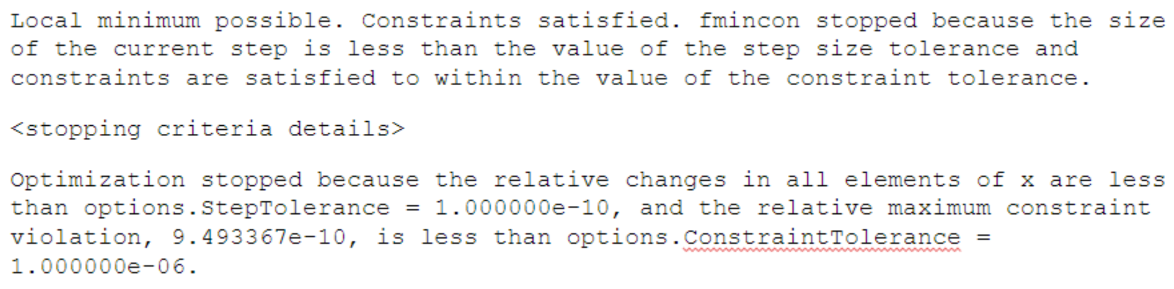
4.3 Solution Analysis

Table 3: Solution Analysis

Termination	Sec. 4.3.1
Local Optimality	Sec. 4.3.2
Global Optimality	Sec. 4.3.3
Uniqueness	Sec. 4.3.4

4.3.1 Termination

`fmincon` reported an exitflag of 2 and terminated the algorithm due to breaching the minimum step size tolerance. The full termination message is shown in Figure 5 and also indicates that the solution satisfies constraints and may be a local minimum. Given the truss design within the airfoil and the prevalence of similar exitflags from our other sensitivity cases, we believe this solution is a local minimum.



```
Local minimum possible. Constraints satisfied. fmincon stopped because the size
of the current step is less than the value of the step size tolerance and
constraints are satisfied to within the value of the constraint tolerance.

<stopping criteria details>

Optimization stopped because the relative changes in all elements of x are less
than options.StepTolerance = 1.000000e-10, and the relative maximum constraint
violation, 9.493367e-10, is less than options.ConstraintTolerance =
1.000000e-06.
```

Figure 5: MATLAB termination message for base case solution with an exitflag of 2

4.3.2 Local Optimality

We believe that the solution is *reasonably close* to a local minima because when we reset the Hessian and initialize a new topology optimization problem using the solution as the initial conditions $\mathbf{x}_0$, there is not a change in the resulting solution. We do not however have definitive evidence that our solution is a KKT point. This is because our solution returned an exitflag of 2 indicating that our step size was below the step tolerance. Ideally, an exitflag of 1 would show that our optimality measure was also below the optimality tolerance and would also indicate a KKT point. Even when we ran `fmincon` with finite difference (Sec. 5.3) instead of providing the replacement gradient, the solution returned an exitflag of 2. This indicates that `fmincon` may be detecting a gradient in the Lagrangian but would require very small step sizes to reach a true local minima. We consider our solution to be very close to a true local minima due to the step size.

4.3.3 Global Optimality

We do not claim to have found a global minimum. Our algorithm arrives at many unique local solutions depending on the initial conditions. Furthermore, our selected solution to the problem is not the solution with the lowest observed objective function value. These "more optimal" solutions, namely the top 3 rows of Fig 7 exhibit checker-boarding that may not translate to a functioning physical design. Therefore we are inclined to search for results using higher values of $r_{\min}$ to induce some amount of filtering and local distribution of densities, rather than a purely binary solution.

4.3.4 Uniqueness

There are many different local minima. In our small parametric study (Fig 7) we saw 27 unique solutions for 27 different sets of input parameters. That said, many of the solutions do exhibit common truss structures or design elements. These solutions are the ones we leaned towards when picking our final design. Certain designs that recur across a variety of input parameters, even across different volume fraction constraints are certainly strong, efficient geometries. Furthermore, we tested the stability of solutions by solving with various input parameters (Sec 5.2 and Fig 6). From this we see that the algorithm is sensitive to initial conditions, producing unique solutions given a different initial $\mathbf{x}_0$, but the algorithm will arrive at the same solution given either the same initial condition ($\mathbf{x}_0 = v_{\mathrm{f}}$, or by setting $\mathbf{x}_0$ equal to the previously found solution. This is encouraging because it shows our algorithm is consistent in finding and remaining at local solutions, even though it exhibits high sensitivity to inputs and with many unique solutions.

4.4 Model Validity

All modeling assumptions are valid at our selected solution. Each element density in the mesh grid is between $0.001 \leq x_i \leq 1$ and all of the active regions have a density $= 1$. The solution appears to make intuitive engineering sense due to the presence of truss structures and supporting connections between the spar locations and mounting points.

4.5 Lagrange Multipliers

Lagrange multipliers for the constraints are provided in Table 4.5 for both the base case solution from Section 4.2 and the finite difference solution from Section 5.3. Since there are 1073 individual constraints for each of the stress, lower bound, and upper bound constraints, we show the sum of the absolute values of the individual Lagrange multipliers instead. We believe the constraints in the base case are active because if they were removed then the optimal solution would likely change.

5 Sensitivity Analysis

5.1 Active Practical Constraints

The practical constraints for this problem are the lower bounds of x_i, volume fraction, and active element constraint. For the base case solution from Section 4.2, all of the Lagrange multipliers are returned as 0 from `fmincon` (Table 4.5). This was unexpected because we can numerically verify that the volume fraction

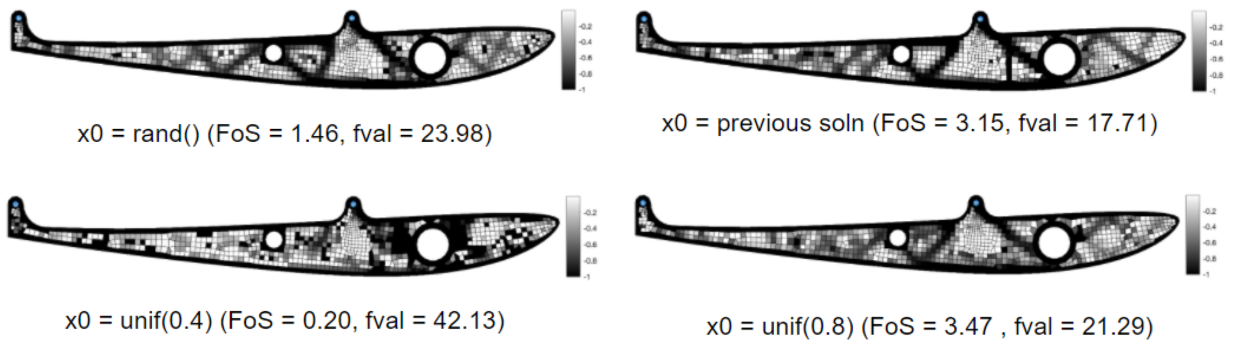
Table 4: Lagrange multipliers for base case solution and finite difference solution

Constraint	Unit	Base Case Multiplier	Active?	Finite Difference Multiplier	Active?
Volume fraction	$[Nm]^{-1}$	0	Yes	-4.76×10^{-4}	Yes
Active elements	$[Nm]^{-1}$	0	Yes	5.53×10^{-4}	Yes
von Mises stress (sum)	m^3	0	No	0	No
x_i lower bound (sum)	$[Nm]^{-1}$	0	Yes	1.56×10^{-4}	Yes
x_i upper bound (sum)	$[Nm]^{-1}$	0	Yes	0.108	Yes

constraint is being met, as well as the active element constraint. Additionally, we can verify that all elements are satisfying the upper and lower bounds on $\mathbf{x}$. To compare, we also listed the Lagrange multipliers from a solution that used finite difference for the gradient calculation, instead of using our replacement gradient. For the finite difference solution, the lower bounds of x_i and the active element constraint returned positive Lagrange multiplier values greater than zero whereas volume fraction returned a negative value. Lagrange multipliers > 0 indicate movement in the optimal solution if the constraints were removed, therefore those constraints are both active and tight. The magnitude of the Lagrange multipliers is small, so the improvement in objective function may be small as well, but it is still possible to find a new and more optimal solution. For a negative Lagrange multiplier, we can conclude that there is additional movement possible in the feasible domain that will improve the objective function. From engineering judgement we suspect volume fraction to be an active constraint even though it has a negative Lagrangian value. This is due to the fact that if the volume fraction constraint was removed then the solution would likely add more volume resulting in a better objective value.

5.2 Parametric Study

Sensitivity analysis was implemented in three phases including the adjustment of initial starting points, variation of key input parameters, and comparison to alternative solutions. The initial decision variable starting points for our solution consisted of each element receiving a value equal to the volume fraction constraint e.g. volume fraction = 0.6. From here we examined a random distribution of starting points along with uniform densities of 0.4 and 0.8. The results are shown in Figure 6 respectfully. We observed that the `fmincon` solution changes based on initial conditions indicating that there are many local minima in this design space. We also noticed that when starting with a uniform density of 0.4 that the algorithm had difficulty finding a solution and required substantial processing time. Furthermore, we examined the solution when starting from previous solution densities and found that the design was similar indicating stability in the optimization algorithm.



x0 = rand() (FoS = 1.46, fval = 23.98)
x0 = previous soln (FoS = 3.15, fval = 17.71)
x0 = unif(0.4) (FoS = 0.20, fval = 42.13)
x0 = unif(0.8) (FoS = 3.47 , fval = 21.29)

Figure 6: Variations in Starting Conditions

The second phase of sensitivity analysis explored simulation results when varying the input parameters of volume fraction, penalty value, and $r_{\min}$. Each parameter was assigned three levels, [0.4, 0.6, 0.8], [1.5, 3, 4.5], and [0.003, 0.006, 0.009] respectfully and used to generate a full enumeration of 27 different cases.

The results are shown in Figure 7 and show vast variability in designs. Our selected design is highlighted in green.

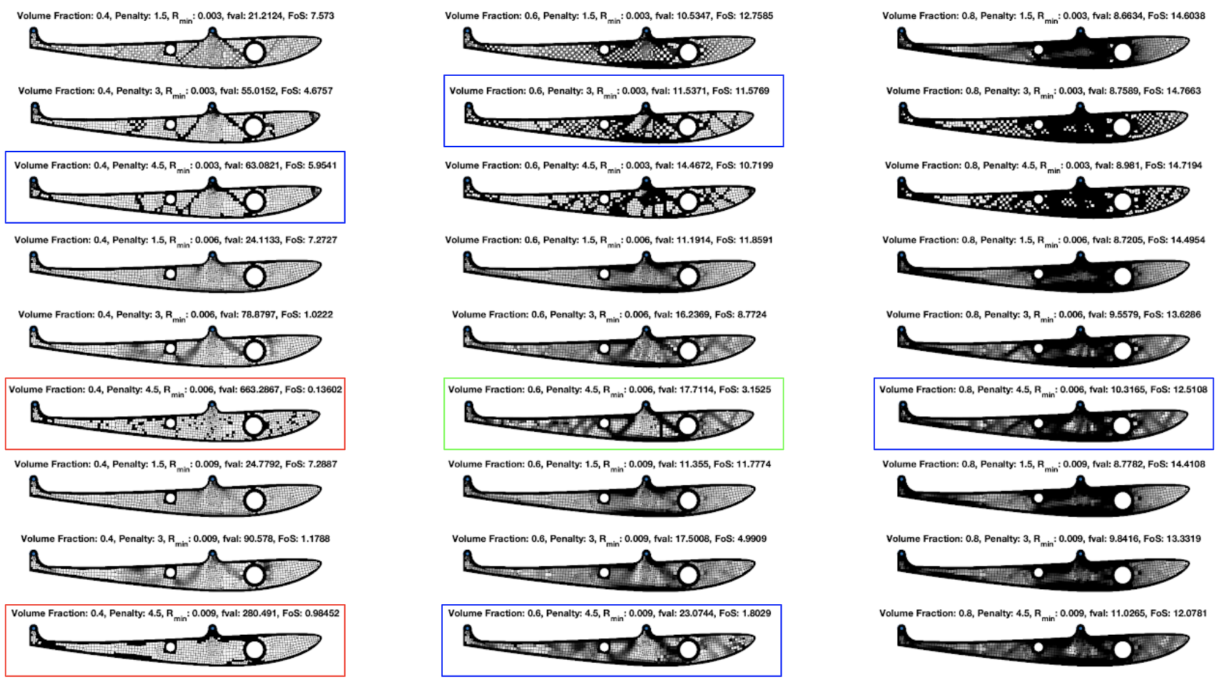

Volume Fraction: 0.4, Penalty: 1.5, R_{min}: 0.003, fval: 21.2124, FoS: 7.573
Volume Fraction: 0.4, Penalty: 3, R_{min}: 0.003, fval: 55.0152, FoS: 4.6757
Volume Fraction: 0.4, Penalty: 4.5, R_{min}: 0.003, fval: 63.0821, FoS: 5.9541
Volume Fraction: 0.4, Penalty: 1.5, R_{min}: 0.006, fval: 24.1133, FoS: 7.2727
Volume Fraction: 0.4, Penalty: 3, R_{min}: 0.006, fval: 78.8797, FoS: 1.0222
Volume Fraction: 0.4, Penalty: 4.5, R_{min}: 0.006, fval: 663.2867, FoS: 0.13602
Volume Fraction: 0.4, Penalty: 1.5, R_{min}: 0.009, fval: 24.7792, FoS: 7.2887
Volume Fraction: 0.4, Penalty: 3, R_{min}: 0.009, fval: 90.578, FoS: 1.1788
Volume Fraction: 0.4, Penalty: 4.5, R_{min}: 0.009, fval: 280.491, FoS: 0.98452
Volume Fraction: 0.6, Penalty: 1.5, R_{min}: 0.003, fval: 10.5347, FoS: 12.7585
Volume Fraction: 0.6, Penalty: 3, R_{min}: 0.003, fval: 11.5371, FoS: 11.5769
Volume Fraction: 0.6, Penalty: 4.5, R_{min}: 0.003, fval: 14.4672, FoS: 10.7199
Volume Fraction: 0.6, Penalty: 1.5, R_{min}: 0.006, fval: 11.1914, FoS: 11.8591
Volume Fraction: 0.6, Penalty: 3, R_{min}: 0.006, fval: 16.2369, FoS: 8.7724
Volume Fraction: 0.6, Penalty: 4.5, R_{min}: 0.006, fval: 17.7114, FoS: 3.1525
Volume Fraction: 0.6, Penalty: 1.5, R_{min}: 0.009, fval: 11.355, FoS: 11.7774
Volume Fraction: 0.6, Penalty: 3, R_{min}: 0.009, fval: 17.5008, FoS: 4.9909
Volume Fraction: 0.6, Penalty: 4.5, R_{min}: 0.009, fval: 23.0744, FoS: 1.8029
Volume Fraction: 0.8, Penalty: 1.5, R_{min}: 0.003, fval: 8.6634, FoS: 14.6038
Volume Fraction: 0.8, Penalty: 3, R_{min}: 0.003, fval: 8.7589, FoS: 14.7663
Volume Fraction: 0.8, Penalty: 4.5, R_{min}: 0.003, fval: 8.981, FoS: 14.7194
Volume Fraction: 0.8, Penalty: 1.5, R_{min}: 0.006, fval: 8.7205, FoS: 14.4954
Volume Fraction: 0.8, Penalty: 3, R_{min}: 0.006, fval: 9.5579, FoS: 13.6286
Volume Fraction: 0.8, Penalty: 4.5, R_{min}: 0.006, fval: 10.3165, FoS: 12.5108
Volume Fraction: 0.8, Penalty: 1.5, R_{min}: 0.009, fval: 8.7782, FoS: 14.4108
Volume Fraction: 0.8, Penalty: 3, R_{min}: 0.009, fval: 9.8416, FoS: 13.3319
Volume Fraction: 0.8, Penalty: 4.5, R_{min}: 0.009, fval: 11.0265, FoS: 12.0781

Figure 7: Parametric Study w/ Variety of Local Minima

There are several key findings from this set of designs. First, not all designs resulted in feasible solution as evident by two designs highlighted in red with volume fraction = 0.4 that violate the maximum stress constraint. Second, there are prevailing design elements in several designs highlighted in blue such as common truss elements and clustering of volume around the spar locations. Third, even though certain designs might predict a high factor of safety, they may not be manufacturable e.g. checkerboard patterns or designs that appears to be empty due to having low density spread across many elements. Finally, the variety of solutions provides further evidence that there are many local minima in this design space and it may be difficult to identify the best solution.

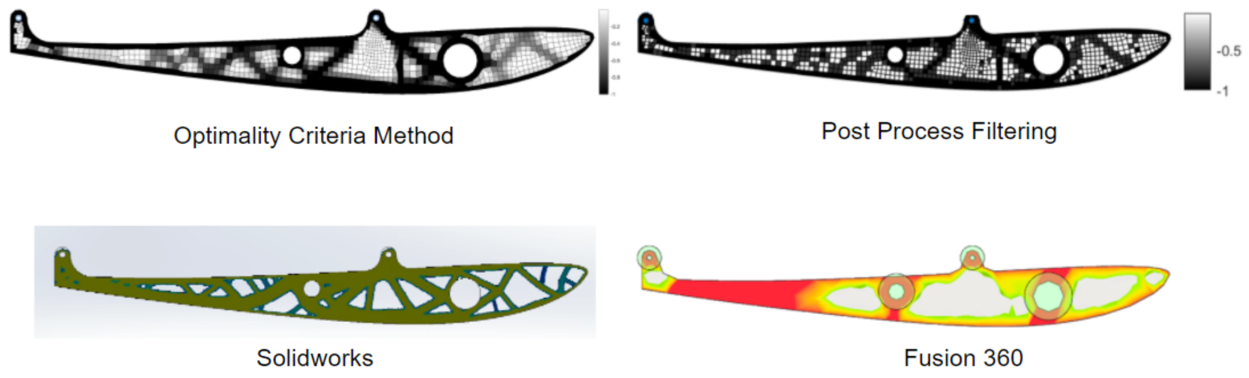


Figure 8: Alternative Solutions

To provide additional insight into the optimal design, the third phase of sensitivity analysis examined design solutions from alternative optimization algorithms. Results from each optimizer are shown in Figure 8.

The first option explored applying an `fmincon` post-processing algorithm that aims to remove element densities above 0.05 but below 0.8 while maintaining a consistent volume fraction. This is intended to create a clear and defined design ready to manufacture. The second alternative applied an optimality criteria optimizer common to topology optimization [7]. This design is slightly different than our selected design and also required less processing time. The final two designs were generated with commercial software packages Solidworks and Fusion 360. These designs contain some similarities, but are also different from our selected design.

5.3 Finite Difference Solution

Instead of using our approximate gradient, we solved the problem using the finite difference approach in `fmincon` as an alternative solution. With the same input parameters as our chosen solution from Sec 4.2: $p = 4.5$, $v_\mathrm{f} = 0.6$, $r_\mathrm{min} = 0.006$, the solver arrived at the solution in Fig 9.

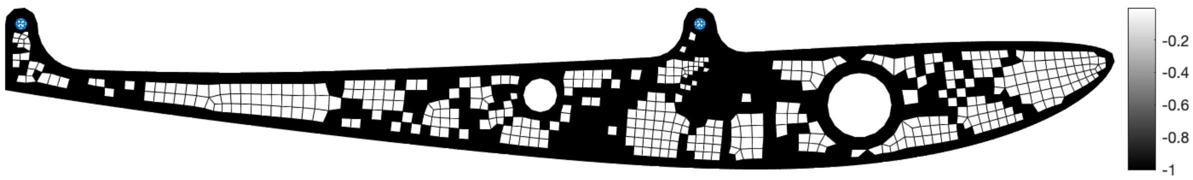


Figure 9: Finite Differences Solution

The solution in Fig 9 has a max stress of 25.7 MPa, for a factor of safety of 10.69 and an fval (compliance value) of 14.7361. According to these quantitative measures, it appears that the finite differences method helped us find a better local solution to the problem. However, if you looked at the geometry in Fig 9, the design exhibits a high degree of checker-boarding and hard to manufacture features. The solver does a fantastic job of pushing elements toward binary solutions, but this also appears to be at the cost of producing an unrealistic geometry. This is where we need to make an engineering decision to not blindly prioritize low objective function values as our sole criterion in selecting a final design.

Furthermore, `fmincon` reports the same termination message and exitflag using finite differences as it does with our approximate gradient. More specifically, the exitflag is 2, and the termination message specifies that the solver stopped because the step size was smaller than the step size tolerance, but at least the constraint tolerance has been satisfied. In summary, finite differences does not produce a compellingly superior solution to the optimization problem to justify the replacement of our approximate gradient method.

6 Conclusions

We employed the SIMP framework and `fmincon` with SQP to minimize the compliance of a finite element airfoil mesh given loading conditions, stress constraints, and volume fraction. Through a parametric study that featured different initial conditions and parameter values, we were able to produce a wide variety of solutions and select a "global" solution based on quantitative and qualitative factors. While none of our `fmincon` solutions indicated a local minimum, we believe the selected design resides sufficiently near a local minima and aligns with engineering design principles. Our final solution provides guidance to where material was most needed on the initial rib design, and therefore shows where material could be taken away. These results could then be incorporated into the design of mounting rib in computer-aided design and then fine-tuned in a commercial FEA program to conform to a desired FOS and provide an engineer with an optimal design.

References

[1] Martin P Bendsøe. "Optimal shape design as a material distribution problem". In: *Structural optimization* 1.4 (1989), pp. 193–202.

[2] Martin Philip Bendsoe and Noboru Kikuchi. "Generating optimal topologies in structural design using a homogenization method". In: (1988).

[3] Allan Peter Engsig-Karup. *QUADPLOT - for plotting 2D quad-meshes.* URL: `https://www.mathworks.com/matlabcentral/fileexchange/32914-quadplot-for-plotting-2d-quad-meshes` (visited on 11/10/2020). Retrieved from MATLAB Central File Exchange.

[4] Erik Holmberg, Bo Torstenfelt, and Anders Klarbring. "Stress constrained topology optimization". In: *Structural and Multidisciplinary Optimization* 48.1 (2013), pp. 33–47.

[5] GP Nikishkov. "Introduction to the finite element method". In: *University of Aizu* (2004), pp. 1–70.

[6] M. Okereke and S. Keates. "Direct Stiffness Method". In: *Finite Element Applications: A Practical Guide to the FEM Process.* Cham: Springer International Publishing, 2018, pp. 47–106. ISBN: 978-3-319-67125-3. DOI: `10.1007/978-3-319-67125-3_3`. URL: `https://doi.org/10.1007/978-3-319-67125-3_3`.

[7] Ole Sigmund. "A 99 line topology optimization code written in Matlab". In: *Structural and multidisciplinary optimization* 21.2 (2001), pp. 120–127.

[8] H.J. Sommer. *polygeom.m.* URL: `https://www.mathworks.com/matlabcentral/fileexchange/319-polygeom-m)` (visited on 11/10/2020). Retrieved from MATLAB Central File Exchange.

Appendix

A1 Code

A1.1 Main Function

```
function [fval,exitflag,loop,maxStress,x,volfrac,lambda,output] = mainMultipleLoads(volfrac,
    penal,rmin)

% clc;
close all; clearvars -except volfrac penal rmin;
% volfrac = 0.6;
% penal = 4.5;
% rmin = 0.006;

load('Airfoil_Uniform_1100_MeshData_v3.mat','ElementList','ElementList_Active_Bool',...
    'NodeList','NodeList_Active','NodeList_Active','NodeList_FixedDOF','
    NodeList_FixedDOF_Bool',...
    'NodeList_Spar1','NodeList_Spar1_Bool','NodeList_Spar2','NodeList_Spar2_Bool','
    NodeList_Surface');
load('LoadingZones.mat','zoneLeadBot','zoneLeadTop','zoneTrailBot','zoneTrailTop','
    zoneMountFront','zoneMountRear');

% INITIALIZE
% Find which elements are active and which nodes are fixed
ElementList_Active_Bool = [ElementList(:,1), any(ismember(ElementList(:,2:end),
    NodeList_Active), 2)];
NodeList_FixedDOF_Bool = [NodeList(:,1), ismember(NodeList(:,1), NodeList_FixedDOF)];

% Material properties
E = 68.9e9;
nu = 0.33;
oy = 276e6;

% Number of Elements and Nodes
nE = size(ElementList,1);
nN = size(NodeList, 1);

% Initialize x0 to all have density of volfrac
x = volfrac*ones(nE, 1);

% Adds fixed active areas to the airfoil, e.g. boundaries of airfoil
% and spars, adds to initial x matrix, added 25 Nov 2020 AJB
for a = 1:nE
    if ElementList_Active_Bool(a,2) == 1
        x(a) = 1;
    end
end

%Matrix for storing and plotting von Mises stresses
stressMat = zeros(nE, 1);

%Pre-compile element stiffness matrices
k_elements = zeros(8,8,nE);
for a = 1:nE
   nodes_i = ElementList(a,[2 3 4 5 2]);
   nodePos_i = NodeList(nodes_i, 2:3)';
   k_elements(:,:, a) = calcStiffness(nodePos_i, E, nu);
end

%Get areas and centroids of elements
[area_e, centroid_e] = calcArea(ElementList,NodeList);

% Setup figures for plotting volume fraction x and vonMises stress
figure('Units', 'normalized', 'Position', [0.5, 0, 0.5, 1]);
ax(1) = subplot(2,1,1);
```

```matlab
plot1 =  meshPlot(ElementList,NodeList, ElementList_Active_Bool, NodeList_FixedDOF_Bool, -x)
    ;
colorbar(ax(1))
title('Volume Fraction')
axis equal; axis tight; axis off;

ax(2) = subplot(2,1,2);
plot2 = meshPlot(ElementList,NodeList, ElementList_Active_Bool, NodeList_FixedDOF_Bool,
    stressMat);
axis equal; axis tight; axis off;
colorbar(ax(2));
title('von Mises Stress')
caxis(ax(2),[0, oy]);

% Initialize while loop counting and termination variables
loop = 0;
change = 1.;
% START ITERATION
while change > 0.01  % Loop until the design does not change significantly. This is typical
    termination criteria in top-opt
    loop = loop + 1;
    xold = x;

    %FE-ANALYSIS
    [U, F]=FE(nE, nN, ElementList, x,penal, k_elements, NodeList_FixedDOF, NodeList_Spar1,
    NodeList_Spar2, NodeList_Surface);

    %Plot boundary conditions once
    if loop == 1
        figure('Units', 'normalized', 'Position', [0,0,0.5,1])
        title("Active Elements and Boundary Conditions")
        subplot(3,1,1); title('Steady-State Load Case');
        subplot(3,1,2); title('Acceleration Load Case');
        subplot(3,1,3); title('Braking Load Case');
        for n = 1:3
            U_plotting = reshape(U(:,n), 2, [])';
            NodeListU = NodeList;
            NodeListU(:,2:3) = NodeListU(:,2:3) + 100*U_plotting;

            subplot(3,1,n); hold on;
            meshPlot(ElementList,NodeList, ElementList_Active_Bool, NodeList_FixedDOF_Bool);
            hold off

            NodePosQuiver1 = NodeList(NodeList_Spar1, 2:3);
            FQuiver1 = [F(NodeList_Spar1*2-1,n), F(NodeList_Spar1*2,n)];

            NodePosQuiver2 = NodeList(NodeList_Spar2, 2:3);
            FQuiver2 = [F(NodeList_Spar2*2-1,n), F(NodeList_Spar2*2,n)];

            NodePosQuiver3 = NodeList(NodeList_Surface, 2:3);
            FQuiver3 = [F(NodeList_Surface*2-1,n), F(NodeList_Surface*2,n)];

            NodePosQuiverTotal = [NodePosQuiver1; NodePosQuiver2; NodePosQuiver3];
            FQuiverTotal = [FQuiver1; FQuiver2; FQuiver3];

            subplot(3,1,n); hold on;
            if n == 2
                quiver(NodePosQuiverTotal(:,1), NodePosQuiverTotal(:,2), FQuiverTotal(:,1),
    FQuiverTotal(:,2),'b', 'LineWidth',2, 'AutoScaleFactor', 0.2)
            else
                quiver(NodePosQuiverTotal(:,1), NodePosQuiverTotal(:,2), FQuiverTotal(:,1),
    FQuiverTotal(:,2),'b', 'LineWidth',2, 'AutoScaleFactor', 0.5)
            end
            hold off
        end
        subplot(3,1,1); axis off; axis equal; axis tight;
        subplot(3,1,2); axis off; axis equal; axis tight;
        subplot(3,1,3); axis off; axis equal; axis tight;
    end

    % DESIGN UPDATE USING FMINCON
    [x,fval,stressMat,exitflag,output,lambda] = solver(x);

    % PRINT RESULTS
    change = max(max(abs(x-xold)));
    disp([' It.: ' sprintf('%4i',loop) ' Obj.: ' sprintf('%10.4f',fval) ...
    ' Vol.: ' sprintf('%6.3f',sum(x.*area_e)/sum(area_e)) ...
    ' maxVMStress: ' sprintf('%6.3e',max(stressMat)) ...
    ' ch.: ' sprintf('%6.3f',change )])

    % PLOT DENSITIES
    set(plot1, 'CData', -x);
    set(plot2, 'CData', stressMat)
    colormap(ax(1), gray);
    colormap(ax(2), parula);

    pause(1e-6);
end

% Post loop thresholding filter to generate a more binary design
%added to show more easily where the final design components should be 6
%Dec 2020
%  x_post_filter = x;
%   criteria = false;
%   while criteria == false
%       for threshold = 0:0.01:1;
%           for i = 1:nE
%             if x_post_filter(i) == 1
%                 x_post_filter(i) = 1;
%             elseif x_post_filter(i) > threshold
%                 x_post_filter(i) = 0.8;
%             else
%                 x_post_filter(i) = 0.05;
%             end
%           end
%             if (x_post_filter'*area_e)/sum(area_e) < volfrac
%                 criteria = true;
%                 x = x_post_filter;
%                 break
%             else
%                 x_post_filter = x;
%             end
%       end
%   end

    %Showing the max VM stress on the figure
    figure(1); subplot(2,1,2);
    str1 = "Yield Stress: " + num2str(oy,'%.3e');
    str2 = "Max VM Stress: " + num2str(max(stressMat),'%.3e');
    annotation('textbox',[0.2 0.05 0.2 0.2],'String',{str1,str2},'FitBoxToText','on','
    HorizontalAlignment','left','VerticalAlignment','middle');

    set(plot1, 'CData', -x);
    colormap(ax(1), gray);

    disp(['Final Obj Func Value: ' sprintf('%f',fval) ' exitflag: ' sprintf('%d',exitflag)])
    output
    maxStress = max(stressMat)
    volfrac

%%%%%%%%% SOLVER %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xnew,fval,stressMat,exitflag,output,lambda] = solver(x)
    % Intialize fmincon
    obj = @objective;
    x0 = x;
    lb = 0.001*ones(size(x));
    ub = 1*ones(size(x));
    nonlcon = @nonlinear;
    A = stress(x);
    b = ones(nE,1)*0;
    % Setting fmincon options
    options = optimoptions(@fmincon,...
    'SpecifyObjectiveGradient',true,...
    'StepTolerance', 1.0000e-10,...
    'MaxFunctionEvaluations',Inf,...
    'MaxIterations',1000,...
    'Algorithm','sqp');
%     % Using fmincon with finite differences
%     options = optimoptions(@fmincon,...
%         'Algorithm','sqp');

    % Calling fmincon
    [xnew,fval,exitflag,output,lambda] = fmincon(obj,x0,A,b,[],[],lb,ub,nonlcon,options);
    exitflag
    fval
    output

    % Calculate Stresses for new design xnew
    stressMat = zeros(nE, 1); %Matrix for storing and plotting vonMises stress
    for i = 1:nE
        %Get Ue
        nodes_i = ElementList(i,[2 3 4 5]);
        edof = reshape([nodes_i*2 - 1; nodes_i*2], [], 1);
        Ue = U(edof, 1);
        %Get vonMises stress
        nodePos_i = NodeList(nodes_i, 2:3)';
        [~, ~, stressVMe] = evalElement(nodePos_i, Ue, E, nu); %Calculate stress
        stressMat(i) = stressVMe*x(i)^(1/2);
    end
end
%%%%%%%%% OBJECTIVE FUNCTION%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [fx,dx] = objective(x)
    % First need to calculate U(x). K(x) is constant
    % Compute U(x)
    [U, F]=FE(nE,nN,ElementList,x,penal,k_elements,NodeList_FixedDOF,NodeList_Spar1,
    NodeList_Spar2,NodeList_Surface);
    % Now Compute c(x) and dc(x)
    c = 0.0;
    dc = zeros(nE,1);
    for m = 1:3
        for i =1:nE
            % First pull correct values for Ue(x)
            nodes_i = ElementList(i,[2 3 4 5]);
            edof = reshape([nodes_i*2 - 1; nodes_i*2], [], 1);
            Ue = U(edof, m);
            % Now compute c(x) and dc(x)
            c = c + x(i)^penal*Ue'*k_elements(:,:,i)*Ue;
            % This isnt actually the gradient. More correct to call it
            % sensitivity?
            dc(i) = dc(i)-penal*x(i)^(penal-1)*Ue'*k_elements(:,:,i)*Ue;
        end
    end
    % Now apply a filter to this sensitivity
    [dc] = check(nE,centroid_e,rmin,x,dc);
    % Assign Outputs
    fx = c;
    dx = dc;
end

%%%%%%%%% LINEAR CONSTRAINT FUNCTION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function [A] = stress(x)
            % Calculate Stresses for new design xnew
            stressMat = zeros(nE, 1); %Matrix for storing and plotting vonMises stress
            for i = 1:nE
                %Get Ue
                nodes_i = ElementList(i,[2 3 4 5]);
                edof = reshape([nodes_i*2 - 1; nodes_i*2], [], 1);
                Ue = U(edof, 1);
                %Get vonMises stress
                nodePos_i = NodeList(nodes_i, 2:3)';
                [~, ~, stressVMe] = evalElement(nodePos_i, Ue, E, nu); %Calculate stress
                stressMat(i) = stressVMe*x(i)^(1/2);
            end
        A = diag(stressMat-oy);
    end

%%%%%%%%%% NONLINEAR CONSTRAINT FUNCTION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [c,ceq] = nonlinear(x)
    c = [];
    ceq(1) = sum(x.*area_e)/sum(area_e)-volfrac; % Volume Fraction Constraint
    ceq(2) = sum(x.*ElementList_Active_Bool(:,2))/sum(ElementList_Active_Bool(:,2)) - 1; %
    Active Elements Constraint
end

%%%%%%%%%% MESH-INDEPENDENCY FILTER %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [dcn]=check(nE,centroid_e, rmin,x,dc)
    dcn=zeros(nE,1);

    for i = 1:nE
        %Find elements within a range rmin
        %Use centroids of elements
        centroid_i = centroid_e(i,:);
        dCentroid = centroid_e - centroid_i;
        distCentroid = sqrt(dCentroid(:,1).^2 + dCentroid(:,2).^2);

        inRange = (distCentroid <= rmin);

        %Do the rest of filtering
        fac = rmin - distCentroid(inRange);
        total = sum(fac);

        dcn(i) = sum(fac.*x(inRange).*dc(inRange));
        dcn(i) = dcn(i)/(x(i)*total);
    end
end

%%%%%%%%%% FE-ANALYSIS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [U, F]=FE(nE, nN, ElementList, x,penal, k_elements, NodeList_FixedDOF,
    NodeList_Spar1, NodeList_Spar2, NodeList_Surface)

    K = sparse(2*nN, 2*nN);
    F = sparse(2*nN,3);
    U = zeros(2*nN,3);

    %Construct stiffness matrix
    for j = 1:nE
        nodes_i = ElementList(j,[2 3 4 5]); %Get nodes for each element
        edof = reshape([nodes_i*2 - 1; nodes_i*2], [], 1); %Get indices of nodes (X and Y)
    in K/F/U matrices
        K(edof, edof) = K(edof,edof) + x(j)^penal*k_elements(:,:,j); %Add to total stiffness
     matrix
    end

    % DEFINE LOADS CASES AND SUPPORTS
    %%%%%%%%%%%%%%%%%%%%%%%%% STEADY STATE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    F(NodeList_Spar1*2,1) = -400; % -Y load on spar 1
    F(NodeList_Spar1*2-1,1) = -80; % -X load on spar 1
    F(NodeList_Spar2*2, 1) = -400; % -Y load on spar 2
    F(NodeList_Spar2*2-1, 1) = -80; % -X load on spar 2

    F(zoneLeadTop*2,1) = -1000; %-Y load on leading edge top
    F(zoneTrailTop*2,1) = -1000; %-Y load on trailing edge top
    F(zoneLeadBot*2,1) = -200; %-Y load on leading edge bottom
    F(zoneTrailBot*2,1) = -200; %-Y load on trailing edge bottom

    F(zoneLeadTop*2-1,1) = -200; %-X load on leading edge top
    F(zoneTrailTop*2-1,1) = -200; %-X load on trailing edge top
    F(zoneLeadBot*2-1,1) = -40; %-X load on leading edge bottom
    F(zoneTrailBot*2-1,1) = -40; %-X load on trailing edge bottom

    %%%%%%%%%%%%%%%%%%%%%%%%%% ACCELERATION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    F(NodeList_Spar1*2,2) = -100; % -Y load on spar 1
    F(NodeList_Spar1*2-1,2) = -10; % -X load on spar 1
    F(NodeList_Spar2*2, 2) = -100; % -Y load on spar 2
    F(NodeList_Spar2*2-1, 2) = -10; % -X load on spar 2

    F(zoneLeadTop*2,2) = -100; %-Y load on leading edge top
    F(zoneTrailTop*2,2) = -100; %-Y load on trailing edge top
    F(zoneLeadBot*2,2) = -100; %-Y load on leading edge bottom
    F(zoneTrailBot*2,2) = -100; %-Y load on trailing edge bottom

    F(zoneLeadTop*2-1,2) = -10; %-X load on leading edge top
    F(zoneTrailTop*2-1,2) = -10; %-X load on trailing edge top
    F(zoneLeadBot*2-1,2) = -10; %-X load on leading edge bottom
    F(zoneTrailBot*2-1,2) = -10; %-X load on trailing edge bottom

    %%%%%%%%%%%%%%%%%%%%%%%%%%%% BRAKING %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    F(NodeList_Spar1*2,3) = -200; % -Y load on spar 1
    F(NodeList_Spar1*2-1,3) = +40; % -X load on spar 1
    F(NodeList_Spar2*2, 3) = -200; % -Y load on spar 2
    F(NodeList_Spar2*2-1, 3) = +40; % -X load on spar 2

    F(zoneLeadTop*2,3) = -800; %-Y load on leading edge top
    F(zoneTrailTop*2,3) = -800; %-Y load on trailing edge top
    F(zoneLeadBot*2,3) = -400; %-Y load on leading edge bottom
    F(zoneTrailBot*2,3) = -400; %-Y load on trailing edge bottom

    F(zoneLeadTop*2-1,3) = 100; %+X load on leading edge top
    F(zoneTrailTop*2-1,3) = 100; %+X load on trailing edge top
    F(zoneLeadBot*2-1,3) = 100; %+X load on leading edge bottom
    F(zoneTrailBot*2-1,3) = 100; %+X load on trailing edge bottom

    fixeddofs   = reshape([NodeList_FixedDOF*2 - 1; NodeList_FixedDOF*2], [], 1);
    alldofs     = [1:2*nN];
    freedofs    = setdiff(alldofs,fixeddofs);
    % SOLVING
    U(freedofs,:) = K(freedofs,freedofs) \ F(freedofs,:);
    U(fixeddofs,:)= 0;
end

%%%%%%%%%% ELEMENT STIFFNESS MATRIX %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function k = calcStiffness(nodePos, E, v)
    % k: individual stiffness matrix for generic quadrilateral element (8x8 matrix)
    % nodePos: X, Y coordinates of nodes (2x4 matrix)...
    %          [X1, X2, X3, X4]
    %          [Y1, Y2, Y3, Y4]
    % Nodes numbered counterclockwise starting with bottom left node
    % 4-------3
    % |       |
    % |       |
    % 1-------2
    %
    % E: Young's modulus (scalar)
    % v: Poisson ratio (scalar)

    x1 = nodePos(1,1);
    x2 = nodePos(1,2);
    x3 = nodePos(1,3);
    x4 = nodePos(1,4);
    y1 = nodePos(2,1);
    y2 = nodePos(2,2);
    y3 = nodePos(2,3);
    y4 = nodePos(2,4);

    % Constituative Matrix for Hookes Law for Plane Stress
    Emat = E/(1-v^2)*[1, v, 0;...
                      v, 1, 0;...
                      0, 0, (1-v)/2];

    % Determinant of Jacobian Matrix for Strain Displacement Matrix
    % (derived with MATLAB Symbolic toolbox for our geometry)
    detJacEq = @(e,n,x1,x2,x3,x4,y1,y2,y3,y4)(x1*y2)/8.0-(x2*y1)/8.0-(x1*y4)/8.0+(x2*y3)
    /8.0-(x3*y2)/8.0+(x4*y1)/8.0+(x3*y4)/8.0-(x4*y3)/8.0-(e*x1*y3)/8.0+(e*x3*y1)/8.0+(e*x1*
    y4)/8.0+(e*x2*y3)/8.0-(e*x3*y2)/8.0-(e*x4*y1)/8.0-(e*x2*y4)/8.0+(e*x4*y2)/8.0-(n*x1*y2)
    /8.0+(n*x2*y1)/8.0+(n*x1*y3)/8.0-(n*x3*y1)/8.0-(n*x2*y4)/8.0+(n*x4*y2)/8.0+(n*x3*y4)
    /8.0-(n*x4*y3)/8.0;

    % Strain Displacement Matrix
    % (derived with MATLAB Symbolic toolbox for our geometry)
    BEq = @(e,n,x1,x2,x3,x4,y1,y2,y3,y4)reshape([((n/4.0-1.0/4.0)*(y1+y2-y3-y4-e*y1+e*y2-e*
    y3+e*y4)*-2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+
    e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+
    n*x3*y4-n*x4*y3)+((e/4.0-1.0/4.0)*(y1-y2-y3+y4-n*y1+n*y2-n*y3+n*y4)*2.0)/(x1*y2-x2*y1-x1
    *y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*
    y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3),0.0,((n
    /4.0-1.0/4.0)*(x1+x2-x3-x4-e*x1+e*x2-e*x3+e*x4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*
    y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n
    *x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)-((e/4.0-1.0/4.0)*(x1-x2-x3+x4-n*
    x1+n*x2-n*x3+n*x4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1
    +e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4
    +n*x4*y2+n*x3*y4-n*x4*y3),0.0,((n/4.0-1.0/4.0)*(x1+x2-x3-x4-e*x1+e*x2-e*x3+e*x4)*2.0)/(
    x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-
    e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)
    -((e/4.0-1.0/4.0)*(x1-x2-x3+x4-n*x1+n*x2-n*x3+n*x4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+
    x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*
    y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3),((n/4.0-1.0/4.0)*(y1+y2-y3-
    y4-e*y1+e*y2-e*y3+e*y4)*-2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e
    *x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n
    *x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)+((e/4.0-1.0/4.0)*(y1-y2-y3+y4-n*y1+n*y2-n*y3+n*y4)*2.0)
    /(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*
    y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*
    y3),((n/4.0-1.0/4.0)*(y1+y2-y3-y4-e*y1+e*y2-e*y3+e*y4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*
    y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*
    x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)-((e/4.0+1.0/4.0)*(y1-y2-
    y3+y4-n*y1+n*y2-n*y3+n*y4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3
    +e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1
    -n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3),0.0,((n/4.0-1.0/4.0)*(x1+x2-x3-x4-e*x1+e*x2-e*x3+e*x4)
    *-2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-
    e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-
    n*x4*y3)+((e/4.0+1.0/4.0)*(x1-x2-x3+x4-n*x1+n*x2-n*x3+n*x4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*
    y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*
    y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3),0.0,((n/4.0-1.0/4.0)
    *(x1+x2-x3-x4-e*x1+e*x2-e*x3+e*x4)*-2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*
    y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*
    y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)+((e/4.0+1.0/4.0)*(x1-x2-x3+x4-n*x1+n*x2-n*x3
    +n*x4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*
    x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*
    x3*y4-n*x4*y3),((n/4.0-1.0/4.0)*(y1+y2-y3-y4-e*y1+e*y2-e*y3+e*y4)*2.0)/(x1*y2-x2*y1-x1*
    y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4
    +e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)-((e
    /4.0+1.0/4.0)*(y1-y2-y3+y4-n*y1+n*y2-n*y3+n*y4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*
    y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n
    *x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3),((n/4.0+1.0/4.0)*(y1+y2-y3-y4-e*
    y1+e*y2-e*y3+e*y4)*-2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*
    y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*
    y4+n*x4*y2+n*x3*y4-n*x4*y3)+((e/4.0+1.0/4.0)*(y1-y2-y3+y4-n*y1+n*y2-n*y3+n*y4)*2.0)/(x1*
    y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*
    x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)
    ,0.0,((n/4.0+1.0/4.0)*(x1+x2-x3-x4-e*x1+e*x2-e*x3+e*x4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3
    *y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*
    x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)-((e/4.0+1.0/4.0)*(x1-x2-
    x3+x4-n*x1+n*x2-n*x3+n*x4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3
    +e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1
    -n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3),0.0,((n/4.0+1.0/4.0)*(x1+x2-x3-x4-e*x1+e*x2-e*x3+e*x4)
    *2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e
    *x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n
    *x4*y3)-((e/4.0+1.0/4.0)*(x1-x2-x3+x4-n*x1+n*x2-n*x3+n*x4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3
    -x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2
    -n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3),((n/4.0+1.0/4.0)*(y1+
    y2-y3-y4-e*y1+e*y2-e*y3+e*y4)*-2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*
    x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*
    x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)+((e/4.0+1.0/4.0)*(y1-y2-y3+y4-n*y1+n*y2-n*y3+n*y4
    )*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-
    e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-
    n*x4*y3),((n/4.0+1.0/4.0)*(y1+y2-y3-y4-e*y1+e*y2-e*y3+e*y4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*
    y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*
    y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)-((e/4.0-1.0/4.0)*(y1
    -y2-y3+y4-n*y1+n*y2-n*y3+n*y4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*
    x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*
    x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3),0.0,((n/4.0+1.0/4.0)*(x1+x2-x3-x4-e*x1+e*x2-e*x3+
    e*x4)*-2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*
    x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*
    x3*y4-n*x4*y3)+((e/4.0-1.0/4.0)*(x1-x2-x3+x4-n*x1+n*x2-n*x3+n*x4)*2.0)/(x1*y2-x2*y1-x1*
    y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4
    +e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3),0.0,((n
    /4.0+1.0/4.0)*(x1+x2-x3-x4-e*x1+e*x2-e*x3+e*x4)*-2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*
    y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n
    *x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)+((e/4.0-1.0/4.0)*(x1-x2-x3+x4-n*
    x1+n*x2-n*x3+n*x4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1
    +e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4
    +n*x4*y2+n*x3*y4-n*x4*y3),((n/4.0+1.0/4.0)*(y1+y2-y3-y4-e*y1+e*y2-e*y3+e*y4)*2.0)/(x1*y2
    -x2*y1-x1*y4+x2*y3-x3*y2+x4*y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*
    y1-e*x2*y4+e*x4*y2-n*x1*y2+n*x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)-((e
    /4.0-1.0/4.0)*(y1-y2-y3+y4-n*y1+n*y2-n*y3+n*y4)*2.0)/(x1*y2-x2*y1-x1*y4+x2*y3-x3*y2+x4*
    y1+x3*y4-x4*y3-e*x1*y3+e*x3*y1+e*x1*y4+e*x2*y3-e*x3*y2-e*x4*y1-e*x2*y4+e*x4*y2-n*x1*y2+n
    *x2*y1+n*x1*y3-n*x3*y1-n*x2*y4+n*x4*y2+n*x3*y4-n*x4*y3)],[3,8]);

    k = 0;
    % Calculate Stiffness Matrix using Gauss Quadrature
    for i = [-1/sqrt(3), 1/sqrt(3)]
        for j = [-1/sqrt(3), 1/sqrt(3)]
            detJac = detJacEq(i, j, x1, x2, x3, x4, y1, y2, y3, y4);
            B = BEq(i, j, x1, x2, x3, x4, y1, y2, y3, y4);

            BEB = B'*Emat*B*detJac;
            k = k + BEB;
        end
    end
end

function [strain, stress, stressVM] = evalElement(nodePos, nodeQ, E, v)
    %
    % nodePos: X, Y coordinates of nodes (2x4 matrix)...
    %          [X1, X2, X3, X4]
    %          [Y1, Y2, Y3, Y4]
    % Nodes numbered counterclockwise starting with bottom left node
    % 4-------3
    % |       |
    % |       |
    % 1-------2
    %
    % nodeQ: X,Y displacements (8x1 matrix)
    %        [dX1; dY1; dX2; dY2; ...]
    %
    % E: Young's modulus (scalar)
    % v: Poisson ratio (scalar)

    x1 = nodePos(1,1);
    x2 = nodePos(1,2);
    x3 = nodePos(1,3);
    x4 = nodePos(1,4);
    y1 = nodePos(2,1);
    y2 = nodePos(2,2);
    y3 = nodePos(2,3);
    y4 = nodePos(2,4);

    % Constituative matrix for ookes law for plane stress
    Emat = E/(1-v^2)*[1, v, 0;...
                      v, 1, 0;...
                      0, 0, (1-v)/2];

    % Strain Displacement Matrix Evaluated at Isoparametric Center of Element
    B = [ (y1 + y2 - y3 - y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 + x3*y4 - x4
    *y3)) - (y1 - y2 - y3 + y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 + x3*y4 -
    x4*y3)), 0, - (y1 + y2 - y3 - y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 + x3
    *y4 - x4*y3)) - (y1 - y2 - y3 + y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 +
    x3*y4 - x4*y3)), 0, (y1 - y2 - y3 + y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*
    y1 + x3*y4 - x4*y3)) - (y1 + y2 - y3 - y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 +
    x4*y1 + x3*y4 - x4*y3)), 0,   (y1 + y2 - y3 - y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3
    *y2 + x4*y1 + x3*y4 - x4*y3)) + (y1 - y2 - y3 + y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 -
    x3*y2 + x4*y1 + x3*y4 - x4*y3)), 0;...
        0, (x1 - x2 - x3 + x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 + x3*y4 -
    x4*y3)) - (x1 + x2 - x3 - x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 + x3*y4
    - x4*y3)), 0,   (x1 + x2 - x3 - x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 +
    x3*y4 - x4*y3)) + (x1 - x2 - x3 + x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1
    + x3*y4 - x4*y3)), 0, (x1 + x2 - x3 - x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4
    *y1 + x3*y4 - x4*y3)) - (x1 - x2 - x3 + x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 +
    x4*y1 + x3*y4 - x4*y3)), 0, - (x1 + x2 - x3 - x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3
    *y2 + x4*y1 + x3*y4 - x4*y3)) - (x1 - x2 - x3 + x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 -
    x3*y2 + x4*y1 + x3*y4 - x4*y3));...
        (x1 - x2 - x3 + x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 + x3*y4 - x4*
    y3)) - (x1 + x2 - x3 - x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 + x3*y4 -
    x4*y3)), (y1 + y2 - y3 - y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 + x3*y4 -
     x4*y3)) - (y1 - y2 - y3 + y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 + x3*y4
     - x4*y3)),   (x1 + x2 - x3 - x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 + x3
    *y4 - x4*y3)) + (x1 - x2 - x3 + x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1 +
    x3*y4 - x4*y3)), - (y1 + y2 - y3 - y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*y1
     + x3*y4 - x4*y3)) - (y1 - y2 - y3 + y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4*
    y1 + x3*y4 - x4*y3)), (x1 + x2 - x3 - x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 + x4
    *y1 + x3*y4 - x4*y3)) - (x1 - x2 - x3 + x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 +
    x4*y1 + x3*y4 - x4*y3)), (y1 - y2 - y3 + y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2 +
     x4*y1 + x3*y4 - x4*y3)) - (y1 + y2 - y3 - y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3*y2
     + x4*y1 + x3*y4 - x4*y3)), - (x1 + x2 - x3 - x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 - x3
    *y2 + x4*y1 + x3*y4 - x4*y3)) - (x1 - x2 - x3 + x4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3 -
    x3*y2 + x4*y1 + x3*y4 - x4*y3)),   (y1 + y2 - y3 - y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*y3
     - x3*y2 + x4*y1 + x3*y4 - x4*y3)) + (y1 - y2 - y3 + y4)/(2*(x1*y2 - x2*y1 - x1*y4 + x2*
    y3 - x3*y2 + x4*y1 + x3*y4 - x4*y3))];

    %Strain-dsiplacement relationship
    strain = B * nodeQ;

    %Hookes Law
    stress = Emat*strain;

    %von-Mises Stress
    stressVM = sqrt(stress(1)^2 - stress(1)*stress(2) + stress(2)^2 + 3*stress(3)^2);
end

function [areas, centroid] = calcArea(ElementList, NodeList)

    %Get nodes for each element
    nodes = ElementList(:,[2 3 4 5 2]);
    nodesX = NodeList(:,2);
    nodesY = NodeList(:,3);

    %Get x and y coordinates for each node for all elements
    xx = nodesX(nodes);
    yy = nodesY(nodes);

    %Calculate lengths of sides of elements
    dx = diff(xx,1,2);
    dy = diff(yy,1,2);
    L = sqrt(dx.^2 + dy.^2);

    %Calculate length of diagonal of element
    dxMid = diff(xx(:, [1,3]), 1, 2);
    dyMid = diff(yy(:, [1,3]), 1, 2);
    LMid = sqrt(dxMid.^2 + dyMid.^2);

    %Use Herons formula for each half of quadrilateral
    s1 = (L(:,1) + L(:,2) + LMid)/2;
    s2 = (L(:,3) + L(:,4) + LMid)/2;

    area1 = sqrt(s1.*(s1-L(:,1)).*(s1-L(:,2)).*(s1-LMid));
    area2 = sqrt(s2.*(s2-L(:,3)).*(s2-L(:,4)).*(s2-LMid));

    areas = area1 + area2;

    %use someone else's modified code to get centroids of each element
    [Geom, ~] = polygeom(xx, yy);
    centroid = Geom(:,2:3);
end
% End of entire function
end
```

A1.2 Subfunction to calculate quadrilateral geometric properties

```
function [ geom, iner ] = polygeom( x, y )
%POLYGEOM Geometry of a planar polygon
%
%   POLYGEOM( X, Y ) returns area, X centroid,
%   Y centroid and perimeter for the planar polygon
%   specified by vertices in vectors X and Y.
%
%   [ GEOM, INER, CPMO ] = POLYGEOM( X, Y ) returns
%   area, centroid, perimeter and area moments of
%   inertia for the polygon.
%   GEOM = [ area   X_cen  Y_cen  perimeter ]
%   INER = [ Ixx    Iyy    Ixy    Iuu    Ivv    Iuv ]
%     u,v are centroidal axes parallel to x,y axes.
%   CPMO = [ I1     ang1   I2     ang2   J ]
%     I1,I2 are centroidal principal moments about axes
%         at angles ang1,ang2.
%     ang1 and ang2 are in radians.
%     J is centroidal polar moment.  J = I1 + I2 = Iuu + Ivv

% H.J. Sommer III - 16.12.09 - tested under MATLAB v9.0
%
% sample data
% x = [ 2.000  0.500  4.830  6.330 ]';
% y = [ 4.000  6.598  9.098  6.500 ]';
% 3x5 test rectangle with long axis at 30 degrees
% area=15, x_cen=3.415, y_cen=6.549, perimeter=16
% Ixx=659.561, Iyy=201.173, Ixy=344.117
% Iuu=16.249, Ivv=26.247, Iuv=8.660
% I1=11.249, ang1=30deg, I2=31.247, ang2=120deg, J=42.496
%
% H.J. Sommer III, Ph.D., Professor of Mechanical Engineering, 337 Leonhard Bldg
% The Pennsylvania State University, University Park, PA  16802
% (814)863-8997  FAX (814)865-9693  hjs1-at-psu.edu  www.mne.psu.edu/sommer/

% begin function POLYGEOM

% check if inputs are same size
if ~isequal( size(x), size(y) )
  error( 'X and Y must be the same size');
end

% temporarily shift data to mean of vertices for improved accuracy
xm = mean(x,2);
ym = mean(y,2);
```

```
x = x - xm;
y = y - ym;

% summations for CCW boundary
xp = x(:, [2:end 1]);
yp = y(:, [2:end 1]);
a = x.*yp - xp.*y;

A = sum(a ,2) /2;
xc = sum((x+xp).*a ,2) /6./A;
yc = sum((y+yp).*a ,2) /6./A;
Ixx = sum((y.*y +y.*yp + yp.*yp).*a, 2) /12;
Iyy = sum((x.*x +x.*xp + xp.*xp).*a, 2) /12;
Ixy = sum((x.*yp +2*x.*y +2*xp.*yp + xp.*y).*a, 2) /24;

dx = xp - x;
dy = yp - y;
P = sum(sqrt(dx.*dx +dy.*dy), 2);

% check for CCW versus CW boundary
if A < 0
  A = -A;
  Ixx = -Ixx;
  Iyy = -Iyy;
  Ixy = -Ixy;
end

% centroidal moments
Iuu = Ixx - A.*yc.*yc;
Ivv = Iyy - A.*xc.*xc;
Iuv = Ixy - A.*xc.*yc;
J = Iuu + Ivv;

% replace mean of vertices
x_cen = xc + xm;
y_cen = yc + ym;
Ixx = Iuu + A.*y_cen.*y_cen;
Iyy = Ivv + A.*x_cen.*x_cen;
Ixy = Iuv + A.*x_cen.*y_cen;

% principal moments and orientation
% I = [ Iuu  -Iuv ;
%      -Iuv   Ivv ];
% [ eig_vec, eig_val ] = eig(I);
% I1 = eig_val(1,1);
% I2 = eig_val(2,2);
% ang1 = atan2( eig_vec(2,1), eig_vec(1,1) );
% ang2 = atan2( eig_vec(2,2), eig_vec(1,2) );

% return values
geom = [ A  x_cen  y_cen  P ];
iner = [ Ixx  Iyy  Ixy  Iuu  Ivv  Iuv ];
% cpmo = [ I1  ang1  I2  ang2  J ];

% bottom of polygeom
```

A1.3 Subfunction to generate plots of 2D mesh

```
function h4 = meshPlot(ElementList, NodeList, ElementList_Active_Bool,
    NodeList_FixedDOF_Bool, fillValue)
    %Setup data to send to quadplot
    %ElementList assumed to be [m x 5]. m is number of elements
    %[element#, node#1, node#2, node#3, node#4]
    % NodePos is assumed to be [n x 3] and sorted in numerical order by node number.
    % n is number of nodes
    % [node#, x#, y#]

    % meshPlot(ElementList,NodeList)
    % meshPlot(ElementList,NodeList, ElementList_Active_Bool, NodeList_FixedDOF_Bool)

    quad = ElementList(:, 2:end);
    x = NodeList(:,2);
    y = NodeList(:,3);

    if (nargin == 2)
        quadplot(quad,x,y)
    elseif (nargin < 5)
        quadplot(quad,x,y, 'k', ElementList_Active_Bool, NodeList_FixedDOF_Bool);
    else
%          quadplot(quad,x,y, 'k', ElementList_Active_Bool, NodeList_FixedDOF_Bool, fillValue
    , clim, colormap)
        h4 = quadplot(quad,x,y, 'k', ElementList_Active_Bool, NodeList_FixedDOF_Bool,
    fillValue);

    end
end

function h4 = quadplot(quad, varargin)
%TRIPLOT Plots a 2D triangulation
%   QUADPLOT(QUAD,X,Y) displays the quadrilaterals defined in the
%   M-by-4 matrix QUAD.  A row of QUAD contains indices into X,Y that
%   define a single quadrilateal. The default line color is blue.
%
%   QUADPLOT(...,COLOR) uses the string COLOR as the line color.
%
%   H = QUADPLOT(...) returns a vector of handles to the displayed
%   quadrilaterals
%
%   QUADPLOT(...,'param','value','param','value'...) allows additional
%   line param/value pairs to be used when creating the plot.
%
%   See also TRISURF, TRIMESH, DELAUNAY, TriRep, DelaunayTri.
%
%   Script code based on copyrighted code from mathworks for TRIPLOT.
%   Allan P. Engsig-Karup, apek@imm.dtu.dk.
error(nargchk(1,inf,nargin,'struct'));
% start = 1;
x = varargin{1};
y = varargin{2};
quads = quad;
if (nargin == 3)
    c = 'blue';
    ElementList_Active_Bool = [];
    NodeList_FixedDOF_Bool = [];
%     start = 3;
else
    c = varargin{3};
    ElementList_Active_Bool = varargin{4};
    NodeList_FixedDOF_Bool = varargin{5};
%     start = 4;
end

if (nargin > 6)
    fillValue = varargin{6};
%     clim = varargin{7};
%     colormap = varargin{8};
end

d = quads(:,[1 2 3 4 1]);

if (nargin > 3)
    e = d(logical(ElementList_Active_Bool(:,2)),:);
end

d = d';

if (nargin > 3)
    e = e';
    f = logical(NodeList_FixedDOF_Bool(:,2));
end

% h = plot(x(d), y(d),c,varargin{start:end});
h = plot(x(d), y(d),c);

if (nargin > 3)
    hold on
    h2 = plot(x(e), y(e),'red');
    h3 = plot(x(f), y(f),'*');
end

h4 = 0;
if (nargin > 6)
    h4 = patch(x(d), y(d), fillValue);
end

axis equal
xlim([-0.025, 0.45])
ylim([-0.1, 0.1])
hold off

if nargout == 1, hh = h; end
end
```

A1.4 Script to run the main loop for repeatedly for parametric study

```
%Run Matrix

function Run_Matrix()
clear all; clc; close all;

    load RunMatrix.mat

      for i = 1:length(RunMatrix.RunOrder)
        tic
        [fval,exitflag,loop,maxStress,xnew,volfrac] = ...
            mainMultipleLoads(RunMatrix.volfrac(i),RunMatrix.penalty(i),RunMatrix.r_min(i));
        i
        time = toc;
        RunMatrix.fval(i,1) = fval;
        RunMatrix.volfrac_final(i,1) = volfrac;
        RunMatrix.exitflag(i,1) = exitflag;
        RunMatrix.loop(i,1) = loop;
        RunMatrix.maxStress(i,1) = maxStress;
        RunMatrix.time(i,1) = time;

        x(:,i) = xnew;
      end

save('multipleLoadsNoTermination')

end
```