Scott Moura
SID 15905638
ME C124
Professor Zohdi
Due: May 10, 2005

Project #2 Report

Consider a base material that is to be doped with particulate additives as shown in Figure 1. Cleary, both the base material and the particulate material have their own set of unique material properties. However, we can establish a desired set of properties for the mixed material, as if to consider the mix homogenized and isotropic. For this project, we use a genetic algorithm to numerically determine the particulate material properties necessary to achieve the desired bulk and shear moduli for the homogenized mixed material.

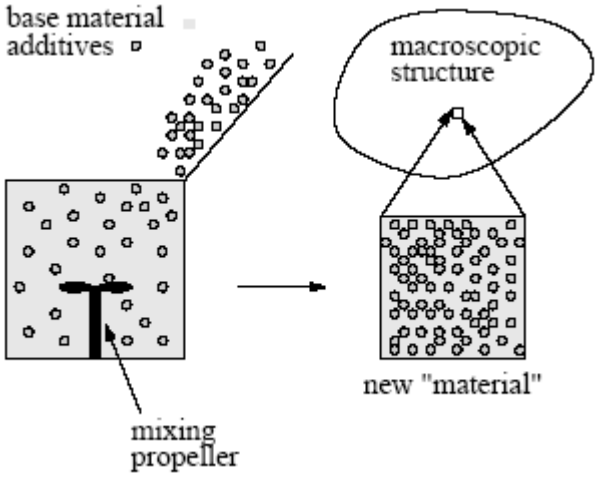

Figure 1: Doping a base material with particulate additives

Convex combinations of the Reuss-Voigt bounds are used to approximate the effective bulk and shear moduli, κ^* and μ^*, of the mixed material. First, consider the Reuss-Voigt bounds for the bulk moduli,

$$\kappa^{*,-} = \left(\frac{v_m}{\kappa_m} - \frac{v_p}{\kappa_p} \right)^{-1} \leq \kappa^* \leq v_m \kappa_m + v_p \kappa_p = \kappa^{*,+}$$

and for the shear moduli,

$$\mu^{*,-} = \left(\frac{v_m}{\mu_m} - \frac{v_p}{\mu_p} \right)^{-1} \leq \mu^* \leq v_m \mu_m + v_p \mu_p = \mu^{*,+}$$

where κ_m and μ_m are the bulk and shear moduli for the matrix material and κ_p and μ_p are the bulk and shear moduli for the particulate material. Also, v_m and v_p are the volume fractions for the matrix and particulate material. To find κ^* and μ^* we take convex combinations of the Reuss-Voigt bounds,

$$\begin{aligned} \kappa^* &\approx \theta\kappa^{*,+} + (1-\theta)\kappa^{*,-} \\ \mu^* &\approx \theta\mu^{*,+} + (1-\theta)\mu^{*,-} \end{aligned} \quad \text{where } 0 \le \theta \le 1$$

For our approximation, we simply use $\theta = 0.5$ for the Reuss-Voigt combination.

Once we have the calculated the effective bulk and shear moduli, κ^* and μ^*, of the mixed material, we try to minimize their difference between the desired bulk and shear moduli, $\kappa^{*,D}$ and $\mu^{*,D}$. To do this, a genetic algorithm is used to minimize the following objective function,

$$\Pi = \left| \frac{\kappa^*}{\kappa^{*,D}} - 1 \right| + \left| \frac{\mu^*}{\mu^{*,D}} - 1 \right|$$

The base material is made out of aluminum with fixed material properties,

$$\begin{aligned} \kappa_m &= 90\text{GPa} \\ \mu_m &= 35\text{GPa} \end{aligned}$$

and the desired material property values for the mixed homogenous material are

$$\begin{aligned} \kappa^{*,D} &= 110\text{GPa} \\ \mu^{*,D} &= 60\text{GPa} \end{aligned}$$

However, the material properties of the particulate material are unknown and must be determined. Thus, the design variables are

$$\Lambda = \left\{ \kappa_p, \mu_p, v_p \right\}$$

To apply the genetic algorithm, we will take random values for the design variables within the following ranges,

$$\kappa_p^- = 90\text{GPa} \le \kappa_p \le \kappa_p^+ = 400\text{GPa}$$
$$\mu_p^- = 40\text{GPa} \le \mu_p \le \mu_p^+ = 200\text{GPa}$$
$$v_p^- = 0.1 \le v_p \le v_p^+ = 0.5$$

We start with a population of 50 random strings within the aforementioned ranges. Once the effective bulk and shear moduli are calculated using convex combinations of Reuss-Voigt bounds, the objective function can be calculated for each of the 50 strings. Since our goal is to minimize the value of Π (get it as close to zero as possible), we can sort the resulting 50 calculations to determine the top ten performing design values. These top ten 'parents' are then 'mated' to produce ten 'offspring' or 'children' design values. The following scheme is used to perform this 'mating' task,

$$\lambda^i = \Phi^{(I)}\overline{\Lambda}^i + (1-\Phi^{(I)})\overline{\Lambda}^{i+1}$$
$$\lambda^{i+1} = \Phi^{(II)}\overline{\Lambda}^i + (1-\Phi^{(II)})\overline{\Lambda}^{i+1}$$

where λ^i are the children design values, $\overline{\Lambda}^i$ are the parent design values, and $\Phi^{(I)}, \Phi^{(II)}$ are random values between zero and one.

At this point, we take the ten parents and ten children and combine them with 30 new random design variables to create a second generation of 50 strings. With these 50 new strings, the objective function is calculated again, sorted, and so forth. This process is run for 100 generations (iterations). At the end of the 100 generations, the top ranked design variables give the bulk and shear moduli and volume fraction for the particulate material that we desire. On top of performing 100 iterations of this genetic algorithm, we perform this entire scheme ten times, thus allowing ten different starting populations. As a result, ten different top performing design values are found for ten different starting populations. The top six values are ranked and shown below,

Rank	**Bulk Modulus** κ_p	**Shear Modulus** μ_p	**Volume Fraction** v_p	Π
1	159.83 GPa	145.01 GPa	0.34243	0.00013
2	158.29 GPa	141.84 GPa	0.35037	0.00034
3	166.25 GPa	155.34 GPa	0.31970	0.00076
4	179.48 GPa	176.43 GPa	0.28182	0.00091
5	192.96 GPa	198.19 GPa	0.24845	0.00300
6	191.73 GPa	190.93 GPa	0.26035	0.00453

It is interesting to note that each run yields different results for the bulk and shear moduli and volume fraction. This must imply that there is an infinite amount of ways to combine the particulate material with the matrix material in

order to achieve the desired properties. From the results above, it is also seen that larger values for the bulk and shear moduli use smaller volume fractions. This seems to make sense as not nearly as much particulate material would be needed to increase the bulk and shear moduli, if the particulate material had high values for each. From these results, we could then find a suitable particulate material that fits these characteristics. Having the top six performing values is useful, just in case we were limited to what possible values we could use. For instance, say we had to use a particulate material that had bulk and shear moduli greater than 160 GPa. In that case, the 4th best design values would be our solution.

In addition, plots for the best performing design values as they ran through the generations are given below in Figure 2. The final value of each plot yields the value of Π given in the table above,

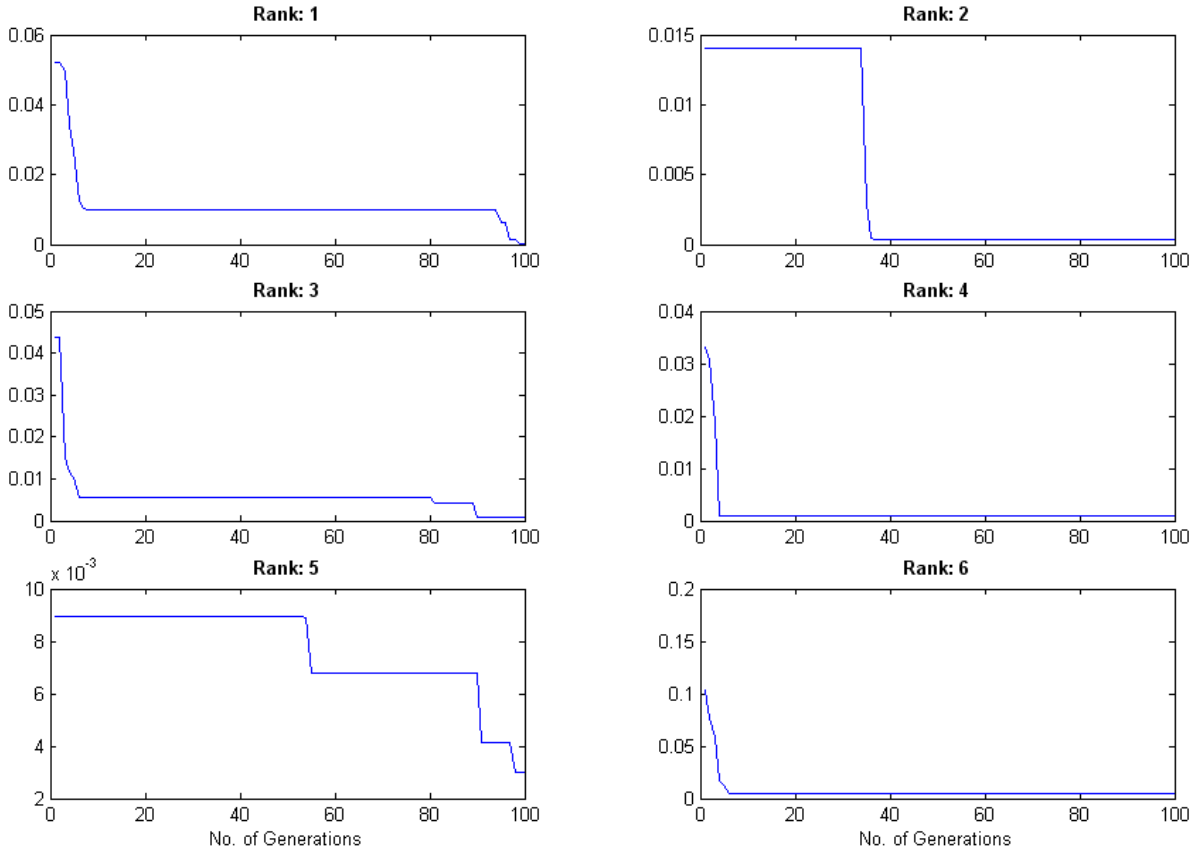

Figure 2: Plots of Top Six Performing Design Values for each Generation

From the plots in Figure 2, it is apparent that some runs immediately find the best performing solutions, like the 4th and 6th place values. However, some runs require nearly all 100 iterations or possibly more to get near the best solution, like the 1st place value. This fact shows that genetic algorithms are best suited when we can take a significantly large number of iterations. Just for this simplified project, we took 1000 iterations across ten different starting populations and there was still movement near the 100th generation. However, the advantage in using genetic algorithms over Newton's method is that we did

not have to take derivatives of the objective function. This fact is key, since in real engineering applications we will probably seldom come across nice continuous differentiable functions. To conclude, genetic algorithms are our best bet when large scale computing is feasible and differentiating objective functions is not. However, if we ever do come across an objective function that is easily differentiable, Newton's method is a quicker and more elegant solution.

```
% Project #2
% genetic.m

% ----------------- %
% Scott Moura       %
% SID 15905638      %
% ME C124           %
% Prof. Zohdi       %
% Due: May 10, 2005 %
% ----------------- %

clear all

% Define constants and parameters
% Desired values for kappa, mu
kappa_D = 110;
mu_D = 60;

% Property bounds for particulate material
kappa_p_min = 90;
kappa_p_max = 400;
mu_p_min = 40;
mu_p_max = 200;
v_p_min = 0.1;
v_p_max = 0.5;

% Properties for matrix material
kappa_m = 90;
mu_m = 35;

% Initialize best1 & best2
best1 = zeros(100,10);
best2 = [];

% Loop for ten different starting populations
for index1 = 1:10

    % Define set of random strings
    d = 50;     % Number of Strings
    random = rand(d,3);
    Lambda(:,1) = kappa_p_min + random(:,1) * (kappa_p_max - kappa_p_min);  % kappa_p
    Lambda(:,2) = mu_p_min + random(:,2) * (mu_p_max - mu_p_min);           % mu_p
    Lambda(:,3) = v_p_min + random(:,3) * (v_p_max - v_p_min);              % v_p

    % Repeat Genetic Algorithm for 100 Generations
    for index2 = 1:100

        kappa_p = Lambda(:,1);
        mu_p = Lambda(:,2);
        v_p = Lambda(:,3);
        v_m = 1 - v_p;

        % Apply Reuss-Voigt bounds
        kappa_min = ((v_m / kappa_m) + (v_p ./ kappa_p)).^-1;
        kappa_max = v_m * kappa_m + v_p .* kappa_p;

        mu_min = ((v_m / mu_m) + (v_p ./ mu_p)).^-1;
```

```
        mu_max = v_m * mu_m + v_p .* mu_p;

        % Use convex approximations for Reuss_Voigt bounds
        theta = 0.5;    % Reuss-Voigt combination value
        kappa = theta*kappa_max + (1 - theta)*kappa_min;
        mu = theta*mu_max + (1 - theta)*mu_min;

        % Calculate the Fitness of the Cost/Objective Function
        Pi = abs(kappa/kappa_D - 1) + abs(mu/mu_D - 1);

        % Rank the Genetic Strings outputed by Pi and save the best
        [sorted_Pi, order_Pi] = sort(Pi);
        best1(index2,index1) = sorted_Pi(1);

        % Keep the Ten Best Parents
        nps = 10;       % Number of Parents
        for i = 1:nps
            parent(i,:) = Lambda(order_Pi(i),:);
        end

        % Mate the Top Ten Parents to Generate Ten Children
        random2 = rand(nps,1);
        for j = 1:2:nps-1
            child(j,:) = random2(j,1) * parent(j,:) + (1 - random2(j,1)) * parent(j+1,:);
            child(j+1,:) = random2(j+1,1) * parent(j,:) + (1 - random2(j+1,1)) * parent(j ↙
+1,:);
        end

        % Generate New Random Strings to Combine with Parents and Children
        clear Lambda_new
        random3 = rand(d-2*nps, 3);

        Lambda_new(:,1) = kappa_p_min + random3(:,1) * (kappa_p_max - kappa_p_min);  % ne ↙
w kappa_p
        Lambda_new(:,2) = mu_p_min + random3(:,2) * (mu_p_max - mu_p_min);           % ne ↙
w mu_p
        Lambda_new(:,3) = v_p_min + random3(:,3) * (v_p_max - v_p_min);              % ne ↙
w v_p

        Lambda_new = vertcat(Lambda_new, parent, child);

        % Repeat Genetic Algorithm
        Lambda = Lambda_new;

    end

Lambda_final(index1,:) = Lambda(50,:);

end

% Determine the runs that had the six best performing designs
best2 = best1(100,:);
[sorted_best2, order_best2] = sort(best2);

% Plot objective value for each generation of the six best designs
figure(1)
for k = 1:6
```

```
    subplot(3,2,k)
    plot(1:100,best1(:,order_best2(k)))
    titl = ['\bfRank: ' int2str(k)];
    title(titl)
    if k >= 5
        xlabel('No. of Generations')
    end

    % Output the design variable values for top six
    best_Lambda(k,:) = Lambda_final(order_best2(k),:);

    fprintf('Rank: %2.0f    kappa_p = %6.2f    mu_p = %6.2f    v_p = %6.5f    Pi = %6.5f\ ↙
n',...
        k,best_Lambda(k,1),best_Lambda(k,2),best_Lambda(k,3),sorted_best2(k));
end
```