DIFFERENTIALLY EVOLVED DESIGN OF AN EQUALIZER BEAM EXPOSED TO UNCERTAIN LOADS FOR USE IN GENERAL PURPOSE LIFTING AND HANDLING

Aaron T. Moore
B.S. May 2013, Old Dominion University

A Project Report Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

MECHANICAL ENGINEERING

May 2019

Mechanical and Aerospace Engineering
Old Dominion University
Norfolk, Virginia, USA
2019-04-30

Approved by:
NOTE: This is a draft.
Approvals have not been given.
Dr. Gene Hou (Advisor)

ACKNOWLEDGEMENTS

First and foremost, I would like to acknowledge Dr. Gene Hou for his guidance and comeraderie throughout developing this document. His wisdom and passion for engineering pushed me to develop this study to a point that would not have otherwise been possible.

I would also like to thank my coworkers at Newport News Shipbuilding for all of their help and guidance whenever I've had a question or needed help thinking something through. I would like to specifically thank Dr. Billy Wagner for inspiring me by sharing his passion for finite element analysis and starting me down the path to this paper.

Finally, I would like to thank NASA for choosing to open-source the 1995 edition of NASTRAN, which wound up being a major factor in the success of this project. I would also like to acknowledge Mr. Luca Dall'Olio for his efforts to make the NASTRAN source code buildable on a modern PC.

List of Code Listings

part of the lifted load, lifting heavy loads near the capacity of the cranes in use can require extremely weight-efficient rigging. At the same time, safety considerations require the stress in these components to be kept as low as achievable to maximize the margin of safety in the rigging system. Because these two objectives are often at odds with one another, design of these equalizer beams taking both objectives into account can be a difficult exercise for the designer.

The nature of the competing requirements imposed against the design of equalizer beams makes them suitable for multi-objective optimization. In general, multi-objective optimization is intended to find a series of optimal solutions across a range of values for the two (or more) different design objectives and present them to a designer. For example, in the case of the equalizer beam, multi-objective optimization can present a series of optimal designs that have the lowest stress at numerous different weight values. Using this technique, a designer can evaluate a number of designs at numerous different combinations of the design variables and select a design that best suits the situation.

Normally, even multiobjective optimization is implemented to handle situations where the input parameters (such as input loads) are clearly defined. In the case of lifting beams, the components can be presented with a variety of loads and loading orientations. To address this, two strategies have been investigated to attempt to handle load input variation in the design. One makes use of a variant of Monte Carlo simulation, and the other uses reliability centric methods.

1.1 Objective

This study will seek to develop a method to design an equalizer beam using multi-objective optimization. Furthermore, the study will seek to incorporate treatment of uncertain input loads in the solution method. The development of the solution will focusing on the optimizer design, using relatively simple methods of varying design parameters.

1.2 Problem Description

As mentioned, the aim of this project is to evaluate multi-objective optimization as a design tool for use in the development of designs for equalizer beams. In order to perform this evaluation and provide a "benchmark case" for any comparisons between methods, an example system will be used as a subject for the design. The basic parameters for the solution are presented below.

1.2.1 Example System

The example system to be studied is based on a few basic fixed design parameters. The basic outline of the beam structure is shown in Figure 1.2.1.

Also fixed is the material, which is assumed to be ASTM A36 steel. The material has properties assumed to be:

- Yield Strength: Mean 250MPa, Std. Deviation 32.5 MPa
- Young's Modulus: 200 GPa

(b) For the Stochastic Loads Method, maximize the critical value for the Reliability Index. This is a reliability measure related, but not identical to stress. The Reliability Index is discussed further in section 2.2.2 on page 9.

1.2.4 Constraints

During the MODE optimization process, 1 constraint is applied. The constraint requires the mass of the modeled beam portion to remain under 1 metric ton (1000kg). In order to accomplish this, a fitness penalty method is used which triggers if the inequality:

$$W_{beam} \leq 1000kg$$

is violated.

The above constraint is applied by generating a multiplier based on the degree to which the constraint has been violated. This multiplier is used to multiply the fitness results, ensuring the solutions that violate constraints are the least dominant and are therefore much less likely to be selected to move on to the next generation in favor of more dominant designs.

1.2.5 Mathematical Representation

To summarize the previous two sections mathematically the objective of this design exercise is:

Given a beam represented by fitness functions:

- $W(t_1, t_2, t_3, t_4, t_5)$: Weight of the beam in kg
- $\sigma(t_1, t_2, t_3, t_4, t_5)$: Peak Stress in the material when loaded.
- $\beta(t_1, t_2, t_3, t_4, t_5)$: Reliability index in the material when loaded.

Find:

$$\min\left(W(t_1, t_2, t_3, t_4, t_5), \sigma(t_1, t_2, t_3, t_4, t_5)\right)$$

OR:

$$\min\left(W(t_1, t_2, t_3, t_4, t_5)\right), \max\left(\beta(t_1, t_2, t_3, t_4, t_5)\right)$$

Subject to:

$$W(t_1, t_2, t_3, t_4, t_5) < 1000\text{kg}$$

w Where the design variables are

t_1 = The thickness of the top flange

t_2 = The thickness of the bottom flange

t_3 = The thickness of the web

t_4 = The thickness of a thickened region surrounding the hoist mounting lug

t_5 = The thickness of a thickened region surrounding the load mounting lug.

For additional details on the design parameters listed above, reference section 2.3.1 ~~Fig. 5~~

Mutation

The mutation operator is the first step for each cycle of the optimization loop. This algorithm emulates random mutations in genetic code commonly found in nature. Taking each individual from the vector of parents, a mutated vector of properties is generated. These vectors are known as *trial vectors*. To perform this action, the basic process flow shown below is employed [6]:

```
trial_vector = empty_2d_vector[num_individuals][num_design_vars]
for integer j in [0 ... num_individuals]:
    x_1 = individuals[j]
    x_2 = random_member_of_individuals
    x_3 = random_member_of_individuals
    beta = x.x // (arbitrary amplification factor selected by user)
    for integer i in [0 ... num_design_vars]:
            trial_vector[j][i] = x_1[i] + beta * (x_2[i] - x_3[i])
return trial_vector
```

Listing 1: Pseudo-code for the Mutation Operator [6]

This set of trial vectors is one of the distinguishing facets of Differential Evolution. If the equation on line 8 above is reviewed carefully, it can be seen that the trial vector is different than its associated parent vector by the distance between 2 other random individuals within the solution space. This has the interesting effect of causing the differences to between parent and trial vectors to change based on the condition of the solution. Early in the solution process when the individuals are sparsely spaced across the solution space, the trial individuals tend to spread apart similarly. In later cycles as minima start to become identified, the distance between individuals becomes smaller. This has the effect of making the trial vectors land more closely to their associated parents. This allows Differential evolution to converge relatively quickly once minima start to appear in the solution space [6].

Crossover

The crossover operator combines the parent individuals and their associated trial individuals to make a single set of *child individuals*. It does this using the following general procedure [6]:

```
child_vector = empty_2d_vector[num_individuals][num_design_vars]
for integer j in [0 ... num_individuals]:
    C = x.x // Constant that dictates how often the crossover picks from the
            // trial vector.
    for integer i in [0 ... num_design_vars]:
        rnd = make_random_number()
        if rnd > C
            child_vector[j][i] = trial_vector[j][i]
        else
            child_vector[j][i] = parent_vector[j][i]
```

Then, an operator $P_{a,b}$ can be defined which returns true if C_a dominates C_b. This would take the following form:

$$P_{a,b} = \begin{cases} \text{True when } C_a^i \leq C_b^i \, \forall i \in \{1..n\} \\ \text{False otherwise} \end{cases} \tag{1}$$

Note that the operator $P_{a,b}$ does not necessarily imply the value of $P_{b,a}$. While only $\vec{C}_b$ or $\vec{C}_a$ can be dominant, it is possible that neither is dominant. In this case, both $P_{a,b}$ and $P_{b,a}$ would be false[6].

Implementing Pareto Dominance in DE

In order to extend DE to be used with multi-objective problems, the code presented in this report simply made a slight modification to the selection operator shown in code listing 3:

```
output_vector = empty_2d_vector[num_individuals][num_design_vars]
for integer j in [0 ... num_individuals]:
    //These statements would return arrays of several fitness values.
    f1 = get_fitness(parent_vector[j])
    f2 = get_fitness(child_vector[j])

    if P(f2,f1)
        output_vector[j] = parent_vector[j]
    else
        output_vector[j] = child_vector[j]
return output_vector
```

Listing 4: Modified Selection Operator

Note that the structure of the if condition "defaults" to selecting the parent. The child is only selected if it is dominant. The parent is selected if it is dominant or if neither dominates.

2.2 Random Loads

This study presents two different approaches for performing MODE optimization on finite element models. The key difference in how the two methods work is the choice of how the random loads in the problem are handled. In this section, the two strategies for generating random loads are introduced.

2.2.1 Latin Hypercube Sampling

Latin hypercube sampling is used throughout the code presented here to generate evenly distributed random data. this method of sampling was selected due to its ability to evenly spread out the sampling points throughout the variable's domain. It does this by programmatically ensuring that each value of each input variable is only represented once. It does this by separating the domain of each input variable into a predetermined number of strata, each with an identical probability of containing an arbitrary sample point.

This turns equation 5 into:

$$\sigma' = \sqrt{\frac{3}{2} \cdot (\alpha)} \tag{7}$$

This simplified equation can be derived as shown below:

$$\frac{\partial \sigma'}{\partial P_x} = \sqrt{\frac{3}{2}} \left[\frac{1}{2} \frac{\partial \alpha}{\partial P_x} (\alpha)^{-\frac{1}{2}} \right] \tag{8}$$

$$\frac{\partial^2 \sigma'}{\partial P_x^2} = \sqrt{\frac{3}{2}} \left[\frac{1}{2} \frac{\partial^2 \alpha}{\partial P_x^2} (\alpha)^{-\frac{1}{2}} - \frac{1}{4} \left(\frac{\partial \alpha}{\partial P_x} \right)^2 (\alpha)^{-\frac{3}{2}} \right] \tag{9}$$

$$\frac{\partial \sigma'}{\partial P_y} = \sqrt{\frac{3}{2}} \left[\frac{1}{2} \frac{\partial \alpha}{\partial P_y} (\alpha)^{-\frac{1}{2}} \right] \tag{10}$$

$$\frac{\partial^2 \sigma'}{\partial P_y^2} = \sqrt{\frac{3}{2}} \left[\frac{1}{2} \frac{\partial^2 \alpha}{\partial P_y^2} (\alpha)^{-\frac{1}{2}} - \frac{1}{4} \left(\frac{\partial \alpha}{\partial P_y} \right)^2 (\alpha)^{-\frac{3}{2}} \right] \tag{11}$$

Of course, this introduces derivatives of α as values that must be calculated. In order to calculate these derivatives, the concept and application of the deviator tensor must be investigated further.

The Deviatoric Stress Tensor (or deviator tensor) describes the component of stress that tends to deform an element. It is given in terms of the overall stress tensor σ' as:

$$\sigma_{dev} = \sigma' - \frac{1}{3}\mathrm{tr}(\sigma') \, [\mathbf{I}] \tag{12}$$

For purposes that will become clear later, we can call this operation a matrix operator γ:

$$\gamma(x) = x - \frac{1}{3}\mathrm{tr}(x) \, [\mathbf{I}] \tag{13}$$

$$\sigma_{dev} = \gamma(\sigma')$$

In this study, σ' is constructed from the component response tensors, σ_{Px} and σ_{Py}. These are tensors that correspond to the stresses in the material when a unit load is applied in the x- and y- directions, respectively. These tensors can be used to define σ':

$$\sigma' = \sigma_{Px} \cdot P_x + \sigma_{Py} \cdot P_y$$

Applying equation 13 to the above definition of σ' yields:

$$\gamma(\sigma') = (\sigma_{Px} \cdot P_x + \sigma_{Py} \cdot P_y) - \frac{1}{3}\mathrm{tr}\left(\sigma_{Px} \cdot P_x + \sigma_{Py} \cdot P_y\right) [\mathbf{I}] \tag{14}$$

From here, it is important to remember that taking the trace of a matrix is a distributive operation, as is

With equations 20 through 23 coupled with equations 8 through 11, we have all of the terms needed to construct and solve equations 3 and 4, substituting the equation for σ into Y for both equations. It is then easy to substitute the results of equations 3 and 4 into 2 and determine β. In the code presented with this report, all of these terms are assembled separately and combined. Therefore, the final equation with all substitutions performed will not be shown here.

Validating the derivation for μ_σ and σ_σ

The equations for finding μ_σ and σ_σ were validated by comparing the results from the equation to a Monte Carlo Simulation of the equation system. One of the test systems used was the following states of stress:

The mean and standard deviation of the input loads to the material was assumed to be:

$$\mu_{px} = 25 \qquad \sigma_{px} = 3.25$$

$$\mu_{py} = 150 \qquad \sigma_{py} = 19.5$$

The unit response to stress was assumed to be:

$$U_{xpx} = 2500 \qquad U_{ypx} = 750 \qquad U_{xypx} = 320$$

$$U_{xpy} = 250 \qquad U_{ypy} = 7500 \qquad U_{xypy} = 210$$

The above unit responses can be represented as tensors. These tensors are:

$$\mathbf{U}_{px} = \begin{bmatrix} 2500 & 320 & 0 \\ 320 & 750 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{U}_{py} = \begin{bmatrix} 250 & 210 & 0 \\ 210 & 7500 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Nice

The deviators of these tensors are:

$$\sigma_{devx} = \begin{bmatrix} 1416.67 & 320 & 0 \\ 320 & -333.3 & 0 \\ 0 & 0 & -1083.33 \end{bmatrix} \qquad \sigma_{devy} = \begin{bmatrix} -2333.3 & 210 & 0 \\ 210 & 4916.67 & 0 \\ 0 & 0 & -2583.33 \end{bmatrix}$$

A monte carlo simulation was prepared using the properties above. A sample size of 1.5×10^6 was used for the simulation. Then, code was created based on the equations 3 and 4, as well as 20 to 23. This code was run with the same parameters, and results obtained. The results from these two analyses can be seen in Table 1 below.

Based on this table, it is plainly apparent that the tensor-based Taylor Series Approximation for finding μ_σ and σ_σ is an appropriate approximation method. The code that was developed for the validation is identical to the code in the final product in the main analysis loop.

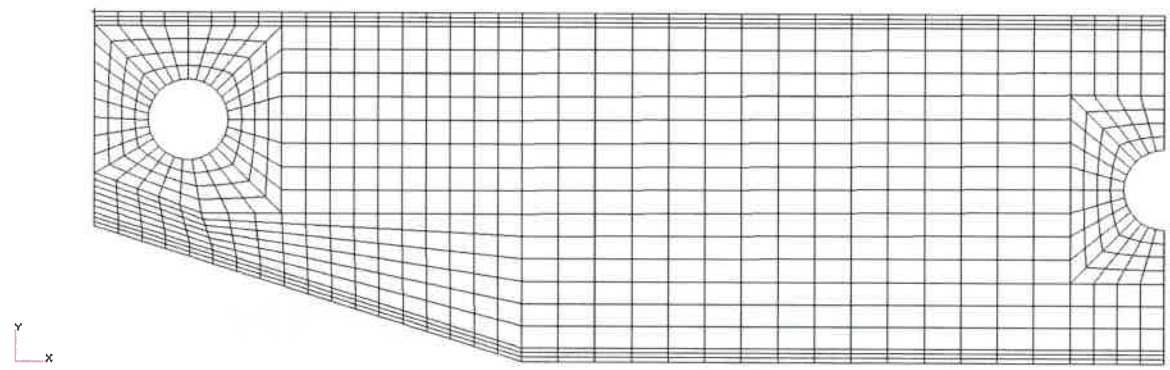

Figure 4: Mesh Geometry

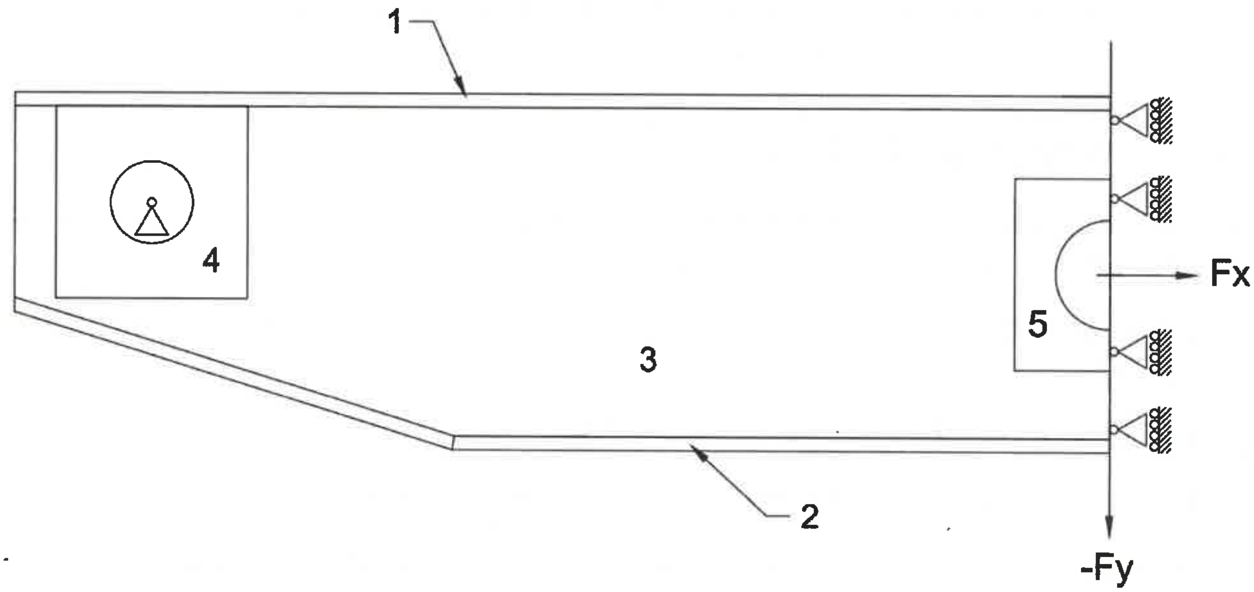

Figure 5: Diagram of Modeled Beam Showing Region Numbers, Force Locations and Constraints.

2. The bottom flange

3. The web

4. A thickened region surrounding the hoist mounting lug

5. A thickened region surrounding the load mounting lug.

Figure 5 shows the named regions as they correspond to portions of the model/drawing.

Also worth noting is that this model is a partial representation of the whole beam. As the example beam in figure 1 shows, the beam features symmetry along the front view. Due to the typical construction of these beams, they also exhibit symmetry along the top view as well. Because of this, the model presented is only one quarter of the complete beam. To account for this, constraints are imposed along the symmetry cut that simulate the remainder of the beam.

3.2.2 msslhs

msslhs is a Python utility that provides latin hypercube sampling in several locations throughout the code presented. This utility is created and maintained by Justin Hughes. It is available from the Mississippi State University CyberDesign Wiki [7].

3.2.3 FEMAP

Pre-processor

3. Aggregating the Pareto Fronts from all load cases to find the overall best designs across all load cases.

Each individual step of this process is outlined in detail below.

Identify Load Cases

In order to find load cases for use, a Latin Hypercube Sampling algorithm is used to select 1000 independent loading conditions within the set of possible load conditions, which is assumed to be a set of normally distributed sample spaces. In this case, the most conservative loadings are those significantly above the mean for the magnitude of the applied force. In order to ensure reliability, the set of load cases to be analyzed is restricted to those load cases more than 1.96 standard deviations above the mean value of the applied load. This ensures that all of the load cases considered are in the top 2.5% of the sample space. This will result in a set of approximately 25 load cases to consider.

(1) How many analyses are needed in each generation? 50 x 25 = ?

or repeat this 25 times, one Pareto front for each load case?

Analyze load Cases

For each load case selected, a set of 50 randomly selected candidate designs are generated. These designs are analyzed using Finite Element ~~Analysis to find~~ the maximum stress and design weight. These values are used as fitness functions to rank the designs. This process is repeated using a Multi Objective Differential Evolution algorithm to selectively refine the designs for a set number of generations.

It was 80 on R20?

Aggregation

At this point, the solution algorithm has 25 separate load cases with individual Pareto fronts. The designs that make up each Pareto front are added to a single unified set of designs. From there, they are once again ranked according to dominance in accordance with Equation 1 and a "final" Pareto front is generated. The designs that comprise this "final" Pareto front are considered to be the most desirable designs and are presented to the designer.

4.2.2 Stochastic Loads Approach

The stochastic loads approach is functionally similar to the Aggregated Latin Hypercube Sampling Approach, with a few key differences:

1. Only a single load case is considered.
2. Stress is not directly considered a fitness variable. Instead, the Reliability Index is considered. While the Reliability Index is proportional to stress, it is not a direct measurement. See Section 2.2.2 for details.

The overall process is detailed below.

Load Case

Instead of using multiple load cases, this approach uses a single load case. However, it is specified as a stochastic set of loads instead of discrete loads applied. For example, the load case considered for the Example System is shown in Table 2. Note that this information matches the design parameters given in

CHAPTER 5

SELECTED RESULTS

The results for sample runs of each solution strategy are printed below. Each method was run twice. One set of solutions were designed to solve in approximately 2.5 hours on the reference hardware. The other run was targeted at approximately an hour. Each pareto optimal design is summarized and the design parameters given in tabular form. Also given are the solution statistics for each run, which include resource usage and wall clock time taken to arrive at a solution. After each set of results is given, the design parameters found by each strategy is compared, as well as the solution statistics.

5.1 Long Runs

The following solutions represent example runs of the solvers when given parameters that result in longer run times. These solutions were run with an expected solution time of approximately 2.5 hours.

5.1.1 Stochastic Loads Method

The Stochastic Loads solver was run with the following argument list:

```
python -m pystruct <<data_file>>  -S3 -i80 -g200 --csv
```

Where:

- `-S3`: Select solution 3: Stochastic Loads Method
- `-i80`: Use a population size of 80.
- `-g200`: Use a generation count of 200.
- `--csv`: Generate CSV output for final Pareto Front

Resultant Pareto Front

Table 3 shows the design parameters of the members of the Pareto Front generated by this solution run. The Pareto Front is also given graphically in Figure 6.

Solution Statistics

This solution also was tracked for several computing performance indicators to compare the performance of the algorithm with the other solutions. These statistics are listed in Table 4.

5.1.2 Aggregate LHS Method

This solution was calculated using parameters designed to produce approximately the same wall clock solution time as the Stochastic Loads Long Run. The command line that was used was:

```
python -m pystruct <<data_file>>  -S2 -i80 -g17 --csv
```

Where:

- `-S2`: Select solution 2: Aggregate LHS method
- `-i80`: Use a population size of 80.
- `-g17`: Use a generation count of 17.
- `--csv`: Generate CSV output for final Pareto Front

Resultant Pareto Front

Table 5 shows the design parameters of the members of the Pareto Front generated by this solution run. The Pareto Front is also given graphically in Figure 7.

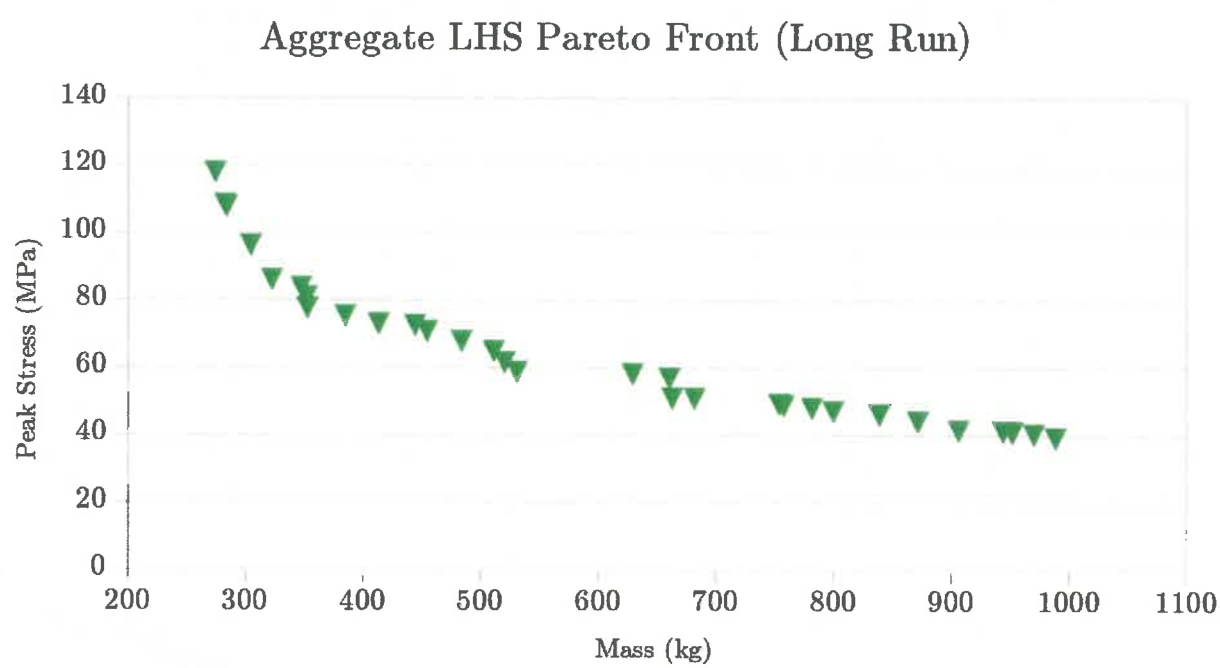

Figure 7: Graph of the Pareto Front generated through Aggregated LHS (Long Run)

Solution Statistics

This solution also was tracked for several computing performance indicators to compare the performance of the algorithm with the other solutions. These statistics are listed in Table 6.

5.2 Short Runs

5.2.1 Stochastic Loads Method

wall clock time?

This solution was run with a targeted execution time of approximately 7-8 minutes. The solution was calculated using the following command line arguments to the solver:

```
python -m pystruct <<data_file>>  -S3 -i30 -g25 --csv
```

Where:

- `-S3`: Select solution 3: Stochastic Loads Method
- `-i30`: Use a population size of 30. ~~Total~~
- `-g25`: Use a generation count of 25.
- `--csv`: Generate CSV output for final Pareto Front

Resultant Pareto Front

Table 7 shows the design parameters of the members of the Pareto Front generated by this solution run. The Pareto Front is also given graphically in Figure 8.

Table 7: Members of the Pareto Front generated through Stochastic Loads (Short Run)

Design Parameters					Fitness Properties	
Top Flange Width	Bottom Flange Width	Web Thickness	Doubler Thickness at Hoist Pin	Doubler Thickness at Load Pin	Reliability Index	Mass
mm	mm	mm	mm	mm	ul	kg
170.039	245.14	71.477	192.605	189.368	6.578	929.701
61.042	223.528	106.182	54.387	103.309	6.651	990.429
248.513	224.849	77.891	30.268	54.893	6.473	816.670
197.478	184.497	17.633	92.008	136.953	5.407	414.890
103.886	16.745	16.36	158.441	185.596	2.562	372.892
230.115	176.491	20.515	21.348	203.78	5.562	419.033
43.425	101.887	53.24	200.759	78.852	5.628	645.181
143.639	222.302	73.671	32.004	87.213	6.437	757.066

Solution Statistics

This solution also was tracked for several computing performance indicators to compare the performance of the algorithm with the other solutions. These statistics are listed in Table 8.

Table 8: Solution Statistics for Stochastic Loads (Short Run)

Total Generations Computed	25
Average Time Per Generation (sec)	18.1
Total Wall Clock Time (sec)	453

Total # of Analysis = 2x30x25

Table 9: Members of the Pareto Front generated through Aggregated LHS (Short Run)

Design Parameters						Fitness Properties	
Parent Load Case	Top Flange Width	Bottom Flange Width	Web Thickness	Doubler Thickness at Hoist Pin	Doubler Thickness at Load Pin	Peak Stress	Mass
	mm	mm	mm	mm	mm	MPa	kg
5	129.488	178.7	50.012	64.189	66.968	65.185	574.583
6	49.705	155.799	78.425	32.802	132.61	52.474	747.643
6	151.264	247.119	22.268	102.256	62.329	73.893	432.479
6	99.534	146.74	64.76	52.983	244.566	53.820	725.557
7	142.734	225.029	57.877	144.821	224.451	49.166	786.819
8	74.907	224.589	54.305	48.5	59.581	61.755	587.581
10	215.954	234.061	88.335	55.669	231.921	38.577	979.637
11	140.513	241.439	18.337	53.885	203.955	80.989	418.511
12	178.503	194.828	43.687	147.502	187.016	59.195	669.831
12	206.098	236.439	75.92	91.15	157.782	42.924	880.824
12	160.811	167.923	69.169	134.097	230.611	48.597	849.269
13	81.25	202.425	42.572	109.538	81.68	67.260	551.180
17	23.721	163.915	49.492	31.35	127.582	72.366	522.037
18	140.976	128.319	39.838	34.495	234.181	72.149	530.609
18	102.296	214.843	13.829	47.973	164.341	89.932	337.531
18	49.437	171.257	79.644	147.498	193.402	48.320	878.472
19	54.821	251.784	24.932	117.632	33.267	86.715	415.135
22	52.03	186.161	10.735	57.554	199.908	118.283	305.286

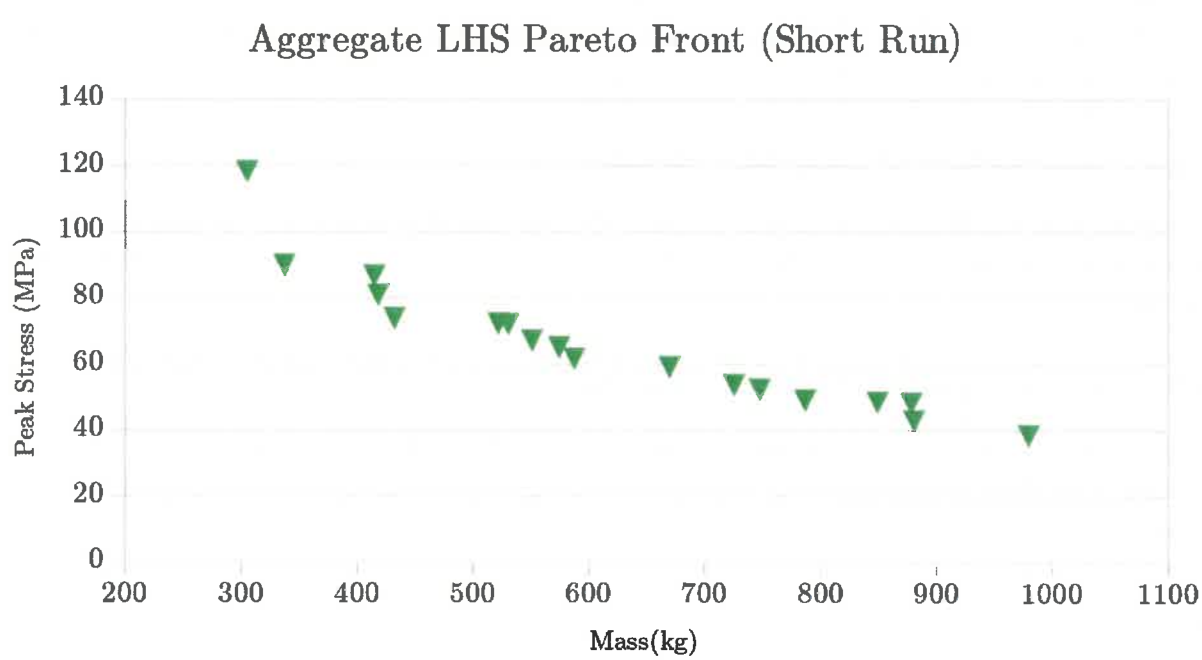

Figure 9: Graph of the Pareto Front generated through Aggregated LHS (Short Run)

Table 10: Solution Statistics for Aggregated LHS (Short Run)

Total Generations Computed	50
Average Time Per Generation (sec)	8.22
Total Wall Clock Time (sec)	411

? Should it be 2 ?

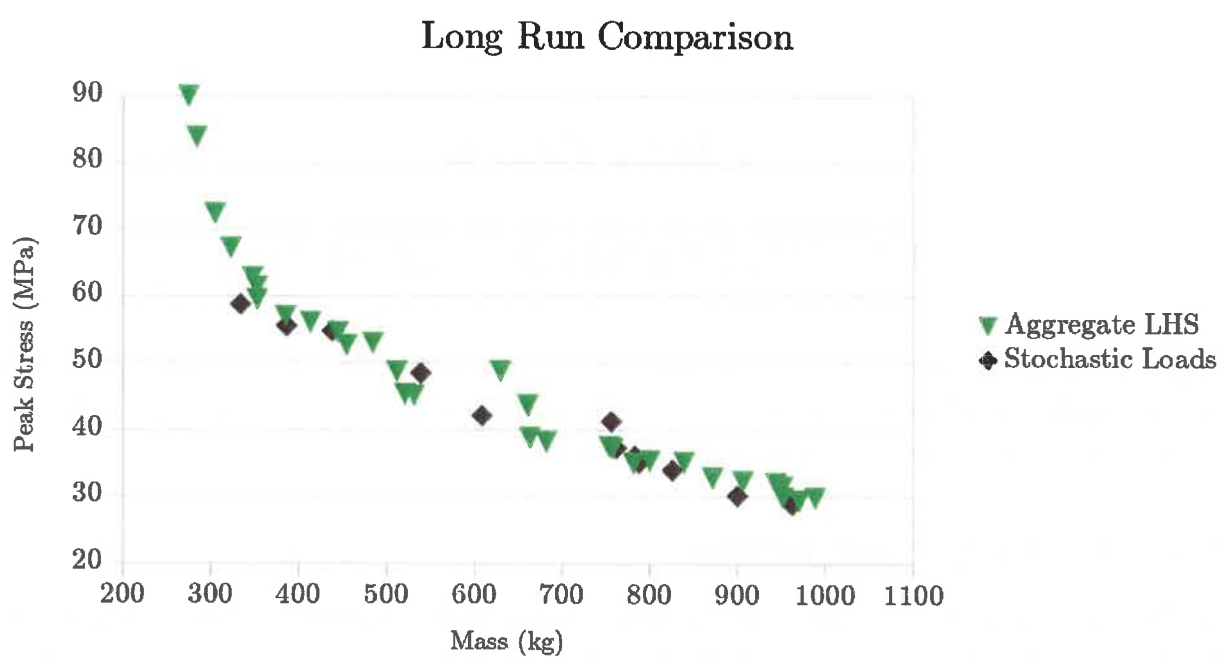

Figure 10: Comparison of the Two Long Run Pareto Plots

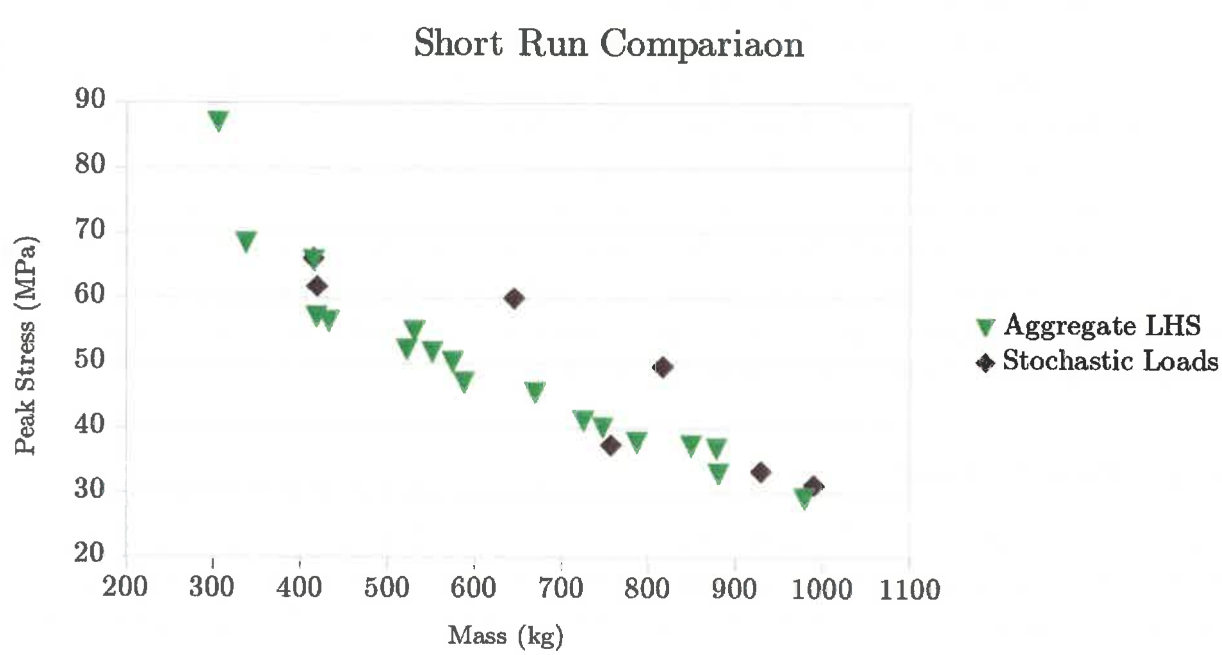

Figure 11: Comparison of the Two Short Run Pareto Plots

method. The efficiency of the Stochastic Loads algroithm can be improved drastically by improving the quality of the implementation of this portion of the code.

CHAPTER 8

APPENDIX A: COMPLETE CODE LISTING