To appear in *Engineering Optimization*
Vol. 00, No. 00, Month 20XX, 1–6

Supplementary Document to the Paper Entitled "Unconventional Optimization for Achieving Well-Informed Design Solutions for an Automobile Industry"

Abhinav Gaur, AKM Khaled Talukder, Kalyanmoy Deb*,
Santosh Tiwari, Simon Xu and Don Jones

(*Received 00 Month 20XX; final version received 00 Month 20XX*)

1. Introduction

In this supplementary document, a detail description of some detail algorithmic implementations of the proposed MOTRAN procedure is presented. Moreover, a brief description of the NSGA-III Deb and Jain (2014) procedure and further results from Stage-3 optimization are provided.

2. Description of MOTRAN Algorithm 1

A procedure to perform a local search on a given feasible solution is described in detail here. Readers are encouraged to refer to Algorithm 1 of the main paper.

- *Input:* The solution to be locally searched is provided as an input: $\boldsymbol{w} = \{w_1, w_2, \ldots, w_n\}$. The lists τ , β, $\boldsymbol{l}$ and $\boldsymbol{u}$ are also provided. A small tolerance `ctol` is used to determine the satisfaction of a constraint is also an input. It is used to relax the lower or upper bound, as the case may be, of each constraint to establish whether a solution is feasible.
- *Output:* Algorithm 1 returns the final feasible locally searched solution $\boldsymbol{v}$ and the updated archive of feasible solutions $\boldsymbol{\Pi}$. Algorithm 1 not only adds the final improved, feasible, and locally searched solution $\boldsymbol{v}$ to the archive $\boldsymbol{\Pi}$, but also adds all the intermediate feasible solutions having a lower y_1 value than that of the original solution $\boldsymbol{w}$.
- *Lines 1-6 :* Line-1 initializes a number of basic variables for the algorithm. Line-2 initializes the solution $\boldsymbol{x}$ with $\boldsymbol{w}$. Variable n keeps a count of number of calls to the program `eval()`. Variable f is a flag which is set to 1, if no feasible descent is possible in a single discrete step of $\pm$ 0.05. Line-3 starts a 'while' loop to search for a solution which is feasible and has lowest response value for y_1 that could be found. Line-4 updates the variable $\boldsymbol{v}$ with the current best-known solution $\boldsymbol{x}$. Lines 5 and 6 are self-explanatory.

*Corresponding author
A. Gaur, A.K.A Talukder and K. Deb are with Computational Optimization and Innovation (COIN) Laboratory, Michigan State University, East Lansing, MI, 48824 USA. e-mail: {gaurabhi, talukde1, kdeb}@msu.edu
S. Tiwari, S. Xu and D. Jones are with the General Motors Vehicle Optimization Group, 30001 Van Dyke, Warren, MI 48093 USA. e-mail: {santosh.tiwari, simon.xu, don.jones}@gm.com

- *Lines 7-8* Line-7 calls the procedure `CV()` given in Algorithm 2 to calculate the list of normalized constraint violations $\boldsymbol{g} = \{g_1, g_2, \ldots, g_{n_c}\}$ and the transformed Jacobian matrix $\boldsymbol{G}$ for a tolerance value of *zero*. The transformation is done in a way such that $\boldsymbol{G}_{(1,q)} = \partial y_1 / \partial x_q$ and $\boldsymbol{G}_{(p,q)} = \partial g_p / \partial x_q$, where $p \in \{2, 3, \ldots, n_r\}$ and $q \in \{1, 2, \ldots, n_v\}$. Here g_i refers to the normalized constraint violation value within some predefined tolerance corresponding to response y_{i+1}. Constraints are modified so that $g_i(\mathbf{x}) > 0$ means that the solution $\mathbf{x}$ does not satisfy the constraint. Line-8 increments i-th variable by δ_i in steps of ± 0.05.
- *Lines 9-16* In the for-loop, the jump in a variable is decided depending upon the values of partial derivative $G_{(1,i)} = \partial y_1 / \partial x_i$ and the bounds of the variable x_i. The parameter δ_i in x_i is set to 0.05 or -0.05, if it has the possibility of reducing the value of y_1 (that is, if $G_{1,i} < 0$, or $G_{1,i} > 0$, respectively) without violating the variable bounds $l_i \leq x_i \leq u_i$. Otherwise, δ_i is set to zero.
- *Lines 17-18* Line-17 checks if no jumps are possible for $\boldsymbol{x}$ to reduce y_1 without violating the corresponding variable bounds and consequently breaks the *while* loop of Line-3 by setting flag f to one. Otherwise, the algorithm continues.
- *Lines 19-21* A jump matrix $\boldsymbol{\Delta}$ is created from $\boldsymbol{\delta}$ such that $\boldsymbol{\Delta}_{(p,q)} = \delta_q$ where $p \in \{1, 2, \ldots, n_r\}$ and $q \in \{1, 2, \ldots, n\}$. Line-21 calculates the *Hadamard product* of transformed Jacobian $\boldsymbol{G}$ and jump matrix $\boldsymbol{\Delta}$ and stores it in the matrix $\boldsymbol{\Theta}$. $\Theta_{(1,q)}$ refers to the possible change in y_1 because of δ_q change in x_q of $\boldsymbol{x}$. Similarly, $\Theta_{(p,q)}$ refers to the possible change in normalized constraint g_{p-1} because of jump δ_q in variable x_q of $\boldsymbol{x}$. Here, $p \in \{2, 3, \ldots, n_r\}$ and $q \in \{1, 2, \ldots, n\}$.
- *Lines 22-31* Line-22 initializes the list $\boldsymbol{\xi} = \{\xi_1, \xi_2, \ldots, \xi_{n_v}\}$ with zeros. In lines 23-25, the variable ξ_i stores the possible value of average normalized constraint violation over all constraints due to a jump of δ_i in variable x_i of $\boldsymbol{x}$. Lines 26-31 find the index i^* of variable jump δ_i in x_i that corresponds to having the maximum potential of reducing response value y_1 by an amount of θ^*, while still being feasible.
- *Lines 32-33* Lines 32-33 terminate the procedure, if there is no i^* in $\{1, 2, \ldots, n_v\}$ with a corresponding $\theta^* < 0$. If a jump with a potential to reduce y_1 while maintaining feasibility is found, then the algorithm continues.
- *Lines 34-end* Line 35-36 create a new variable $\hat{\boldsymbol{x}}$ from $\boldsymbol{x}$ by adding a jump of δ_i to variable x_i of $\boldsymbol{x}$. Line 37-38 call the evaluator to make an actual evaluation on $\hat{\boldsymbol{x}}$ to obtain corresponding responses $\hat{\boldsymbol{y}}$ and increases the total function evaluations H by one. Line-39 calls the procedure `CV()` (Algorithm 2) to obtain corresponding normalized constraint violation values $\hat{\boldsymbol{g}}$ within the desired tolerance `ctol`. Line-40 checks if the response $\hat{y}_1$ of solution $\hat{\boldsymbol{x}}$ is indeed smaller than y_1 of solution $\boldsymbol{x}$ and whether the solution $\hat{\boldsymbol{x}}$ has a zero total constraint violation. If both conditions are found to be true, then solution $\boldsymbol{x}$ is updated in Line-41 and subsequently added to the archive $\boldsymbol{\Pi}$. If the aforementioned conditions are not met, then flag variable f is set to one to break the *while* loop of line-3. Subsequently, the algorithm returns the best solution found, $\boldsymbol{v}$, and the updated archive $\boldsymbol{\Pi}$ to the calling program.

3. Description of Algorithm 2

A procedure to calculate normalized constraint violation and the normalized Jacobian from the response values of a solution is described here. Readers are encouraged to refer to Algorithm 2 of the main paper.

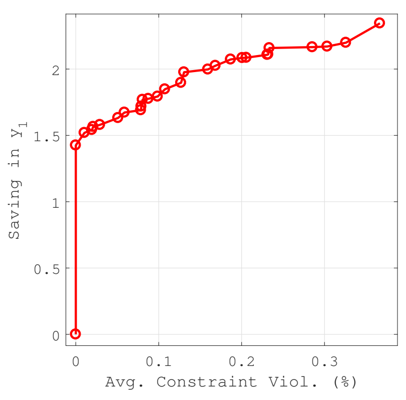
- *Input:* For some solution, say $\boldsymbol{x} = \{x_1, x_2, \ldots, x_{n_v}\}$, the corresponding response vector $\boldsymbol{y} = \{y_1, y_2, \ldots, y_{n_r}\}$ and the Jacobian matrix $\boldsymbol{J}$ (size $n_r \times n_v$) are provided to Algorithm 2 as input. Recall that $n_v = 145$ is the number of variables and $n_r = 147$

is the number of responses. Furthermore, the parameters $\boldsymbol{\tau}$, $\boldsymbol{\beta}$ and `ctol` are same as described in Section 2.

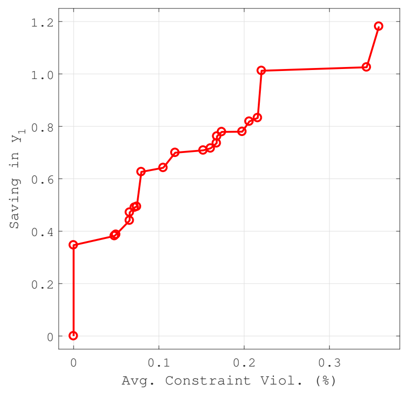
- *Output:* Algorithm 2 returns the normalized constraint violation $\boldsymbol{g}$ and normalized Jacobian $\boldsymbol{G}$ as output.
- *Lines 1-3:* Line-1 initializes a number of basic variables for the algorithm. Line-2 initializes the normalized constraint violation vector $\boldsymbol{g}$ with $\mathbf{0}$ and normalized Jacobian $\boldsymbol{G}$ with Jacobain $\boldsymbol{J}$ respectively. Line-3 is the beginning of a `for` loop that calculates different components of $\boldsymbol{g}$ and $\boldsymbol{G}$ for each of the n_c constraints.
- *Lines 4-8:* For some i^{th} constraint, these lines check if the constraint has an upper bound or a lower bound. If the constraint has an upper bound, the flag m is set to 1, else to -1.
- *Lines 9-13:* Line-9 then checks if the absolute value of the i^{th} constraint bound is greater than one. If it is, then the value of the i^{th} constraint is normalized as shown in Line-10. Note that the i^{th} constraint value is based on $(i+1)^{th}$ response value y_{i+1} as response vector y has 147 components, the first of which is just an objective and no constraint is based off of it. Line-11 to Line-13 represent a `for` loop to normalize the i^{th} row values of Jacobian $\boldsymbol{J}$.
- *Lines 14-19:* If the absolute value of the i^{th} constraint bound is ≤ 1, then constraint and Jacobian normalization is as shown in Line-15 to Line-18.
- *Lines 20-25:* Line-20 subtracts the allowed tolerance `ctol` on each normalized constraint value. Line-21 to Line-23 assign the normalized constraint value of g_i to be 0 if i^{th} constraint value is less than zero, that is, it is feasible within the specified tolerance limit. Line-24 marks the end of the `for` loop started in Line-3. Line-25 then returns the normalized constraint violation $\boldsymbol{g}$ and normalized Jacobian $\boldsymbol{G}$.

4. NSGA-III Procedure

The NSGA-III algorithm (Deb and Jain 2014) starts with a parent population of randomly created solutions. The size of the population can be set as equal to the number of desired Pareto-optimal solutions. To begin with, a set of well-distributed set of reference directions (Das and Dennis 1998) originating from the origin are pre-specified in the first coordinate system of the m-dimensional objective space (m is the number of objectives). One generation of NSGA-III procedure works as follows. An offspring population of the same size as the parent population size is created by using algorithm's operators, which can be customized using problem information to have a more efficient application. The populations are combined to form a merged population. First, all objective values of the merged population members are normalized so that the resulting non-dominated solutions lie within [0,1]. This allows the population members to be compared with pre-specified reference lines. Each population member is associated with its nearest reference line and in turn, all associated members of each reference line are identified and grouped together. The merged population is then ranked according to increasing levels of constraint non-domination (Deb 2001; Deb et al. 2002) using all m objectives and constraints. Since level 1 solutions are infinitely better than subsequent levels, all solutions are accepted level-wise starting from level 1, until no more levels can be accepted to fill the new population slots. Note that only half of the merged population can be accepted to make the overall process a systematic algorithm. The last level of solutions which could not be accepted as a whole to meet the population size requirement are then considered for the final niching operator to select a well-diversed set of remaining population members. For each reference line, its nearest associated member is chosen for this purpose. It is important to mention that in large-objective problems, almost all merged population members lie on level 1 of non-domination after a few generations, and the



(a)



(b)

Figure 1.: Saving in objective y_1 versus average constraint violation shown for the two niched solutions.

final niching operator becomes the main selection operator of the NSGA-III procedure. Due to the emphasis of non-dominated solutions and well-diversified solutions through individual picking of closely adhered solutions of a set of well-distributed reference lines, NSGA-III is able to converge with a good distribution of non-dominated and trade-off solutions at the end of a simulation run.

5. Further Stage-3 Results

For the two solutions having $y_1 = 168.78$ and 168.93 (obtained from the Stage-1 multi-objective optimization), Stage-3 is applied one at a time. Figures 1 show the respective bi-objective trade-off solutions. Interestingly, in both cases, a large gain in y_1-objective is possible without any sacrifice of the constraint satisfaction. Thus, in addition to finding trade-off solutions between constraint relaxation and objective gain, Stage-3 also acts as an additional local search operator in the vicinity of the chosen Stage-1 solution. Similar trade-off behavior in both these cases can also be observed from the figures.

6. Conclusions

Along with the description and results presented in the main paper, this supplementary document provides more information and results in favor of the proposed MOTRAN procedure. It is ready to be applied to other similar engineering design optimization problems, primarily for exploring different feasible designs so that a more informative and holistic optimal design can be achieved.

Acknowledgment

This research was supported by General Motors. The views presented here do not necessarily reflect those of the sponsor whose support is gratefully acknowledged.

References

Das, I., and J.E. Dennis. 1998. "Normal-Boundary Intersection: A new method for generating the Pareto surface in nonlinear multicriteria optimization problems." *SIAM Journal of Optimization* 8 (3): 631–657.

Deb, K. 2001. *Multi-objective optimization using evolutionary algorithms*. Chichester, UK: Wiley.

Deb, K., S. Agrawal, A. Pratap, and T. Meyarivan. 2002. "A fast and Elitist multi-objective Genetic Algorithm: NSGA-II." *IEEE Transactions on Evolutionary Computation* 6 (2): 182–197.

Deb, K., and H. Jain. 2014. "An Evolutionary Many-Objective Optimization Algorithm Using Reference-point Based Non-dominated Sorting Approach, Part I: Solving Problems with Box Constraints." *IEEE Transactions on Evolutionary Computation* 18 (4): 577–601.