# TherMaG: Engineering Design of Thermal-Magnetic Generator with Multidisciplinary Design Optimization

Will Hintlian*, Hanfeng Zhai†, Mads Peter Berg‡

*Sibley School of Mechanical and Aerospace Engineering*

*Applied and Engineering Physics*

*Cornell University*

November 20, 2021

## 1 Scaling

### 1.1 Cost

As referred back to the original optimization problem, we still focus on the `cost` module as for the scaling problem with gradient based method `fmincon`.

We adopt the `sqp` algorithm on the **cost** module with the five input design variables $(0.05; 0.1; 0.05; 0.05; 0.05)$ for running the optimization; the objective of cost module, the geometric constraint and the main optimization program are same as our previous work (*as attached in the Appendix*). Note that the upper and lower bounds are $(.45, .45, .45, .45, .45)$ and $(.05, .05, .05, .05, .05)$. By running the optimization we generate the following output:

```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.
```

*Email: wth42@cornell.edu
†Email: hz253@cornell.edu
‡Email: mpb99@cornell.edu

```
<stopping criteria details>
No scaling: The best solution is efficiency = 0.34027460, corresponding to a design vector
    of:

    0.0010
    0.0020
    0.0010
    0.0010
    0.0010


This solution had the following simulation output info:

         iterations: 2
          funcCount: 18
          algorithm: 'sqp'
            message: ' Local minimum found that satisfies the constraints.  Optimization
    completed because the objective function is non-decreasing in  feasible directions, to
    within the value of the optimality tolerance, and constraints are satisfied to within
    the value of the constraint tolerance.  <stopping criteria details>  Optimization
    completed: The relative first-order optimality measure, 2.228358e-11, is less than
    options.OptimalityTolerance = 1.000000e-06, and the relative maximum constraint
    violation, 0.000000e+00, is less than options.ConstraintTolerance = 1.000000e-06.  '
    constrviolation: 0
           stepsize: 1.4867e-17
       lssteplength: 1
      firstorderopt: 7.4506e-09
       bestfeasible: [1x1 struct]

The unscaled hessian is:

   1.0e+05 *

    0.0005    0.0003    0.0142    0.0149    0.0291
    0.0003    0.0001    0.0071    0.0075    0.0145
    0.0142    0.0071    0.3895    0.4104    0.7998
    0.0149    0.0075    0.4104    0.4325    0.8428
    0.0291    0.0145    0.7998    0.8428    1.6427

The unscaled condition number is 7.144017e+05
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

```

```
<stopping criteria details>
Cost scaling: The best solution is cost = 0.00192677, corresponding to a design vector of:

    0.0010
    0.0020
    0.0010
    0.0010
    0.0010


This solution had the following simulation output info:

         iterations: 2
          funcCount: 18
          algorithm: 'sqp'
            message: ' Local minimum found that satisfies the constraints.  Optimization
    completed because the objective function is non-decreasing in  feasible directions, to
    within the value of the optimality tolerance, and constraints are satisfied to within
    the value of the constraint tolerance.  <stopping criteria details>  Optimization
    completed: The relative first-order optimality measure, 1.332268e-15, is less than
    options.OptimalityTolerance = 1.000000e-06, and the relative maximum constraint
    violation, 0.000000e+00, is less than options.ConstraintTolerance = 1.000000e-06.  '
    constrviolation: 0
           stepsize: 9.8027e-17
       lssteplength: 1
      firstorderopt: 1.3323e-15
       bestfeasible: [1x1 struct]

The scaled hessian is:

   25.9973   12.3112    2.0591    2.1765    4.3607
   12.3112    6.7806    0.8421    0.9008    1.9928
    2.0591    0.8421    1.0649    0.0751    0.2650
    2.1765    0.9008    0.0751    1.0859    0.2859
    4.3607    1.9928    0.2650    0.2859    1.6759

The scaled condition number is 6.270862e+01
The scaled optimization found a 0 percent smaller optimal value in the same number of
    iterations and -0 percent fewer function evaluations

The scaled optimization found a solution 87.90 percent faster than the unscaled optimization
    >>
```

Therefore we can write out the unscaled Hessian:

$$\mathbf{H} = \begin{bmatrix} 52.4001 & 25.5125 & 1416.4316 & 1492.5933 & 2909.1499 \\ 25.5125 & 13.3813 & 708.0283 & 746.1091 & 1454.3874 \\ 1416.4316 & 708.0283 & 38945.6567 & 41038.5367 & 79983.3184 \\ 1492.5933 & 746.1091 & 41038.5367 & 43245.9949 & 84283.6567 \\ 2909.1499 & 1454.3874 & 79983.3184 & 84283.6567 & 164268.1001 \end{bmatrix} \tag{1}$$

With simple observation we can deduce that the problem is obviously ill conditioned; A scaling of the problem is hence required.

The five input design variables are $[w_{yk}, h_{yk}, h_{pm}, h_A, w_{gap}]$, to approximate the final form of $\mathcal{O}(\mathbf{H}) \approx 1$; we hope to make each terms of $\mathbf{H} \sim \mathbf{1}$. Therefore, the scaling factor should be $[10^{-1.5}; 10^{-.5}; 10^{-2}; 10^{-2}; 10^{-2.5}]$. Same as our previous settings, the lower and upper bounds of the optimization system are $(0.05, 0.05, 0.05, 0.05, 0.05)$ and $(0.45, 0.45, 0.45, 0.45, 0.45)$, respectively. Applying the factors and scale the new design variables as $[\tilde{w}_{yk}, \tilde{h}_{yk}, \tilde{h}_{pm}, \tilde{h}_A, \tilde{w}_{gap}] = [10^{-1.5} w_{yk}, 10^{-.5} h_{yk}, 110^{-2} h_{pm}, 10^{-2} h_A, 10^{-2.5} w_{gap}]$ and reoptimize the problem (*the new code for the reoptimization are also attached in the Appendix*).

From the output we can deduce that the new Hessian $\tilde{\mathbf{H}}$ is

$$\tilde{\mathbf{H}} = \begin{bmatrix} 25.9973 & 12.3112 & 2.0591 & 2.1765 & 4.3607 \\ 12.3112 & 6.7806 & 0.8421 & 0.9007 & 1.9928 \\ 2.0591 & 0.8421 & 1.0649 & 0.07509 & 0.2650 \\ 2.1765 & 0.9007 & 0.0751 & 1.0859 & 0.2859 \\ 4.3607 & 1.9928 & 0.2649 & 0.2859 & 1.6759 \end{bmatrix} \tag{2}$$

It can be deduced that the problem is no longer ill-conditioned. Also, surprisingly, the scaled and original optimization both land on the same solution for the design variables: $(0.05, 0.05, 0.05, 0.1, 0.05)$.

To estimate the how scaling works for the cost module, we mainly investigate *compute time, condition number, number of function evaluations, and quality of the optimized solution* before and after the scaling.

Considering computation time, we observe that using scaling on the cost function allows the optimization to run 87% faster while performing the same number of function evaluations.

Considering the condition number, we can apply the MATLAB® built-in function `cond()` to compute the condition number of our Hessian, given by

$$\kappa(\mathbf{H}) = ||\mathbf{H}||||\mathbf{H}^{-1}|| \tag{3}$$

Before the scaling the condition number is $\kappa(\mathbf{H}) = 7.144 \times 10^5$; and after the scaling the condition is $\kappa(\tilde{\mathbf{H}}) = 6.271 \times 10$. It is obvious that the condition number dropped significantly after scaling, indicating an effective scaling.

Eventually, when looking at the quality of the optimization, the `First-order optimality` could help us deduce. First-order optimality is a measure of how close a point x is to optimal [4]. Here, in our approach, the objective takes the form

$$\min \texttt{cost}(w_{yk}, h_{yk}, ...) \tag{4}$$

the first-order optimality measure is the infinity norm (meaning maximum absolute value) of

$$\max_i |(\Delta\texttt{cost}(\text{geomtric vars.}))_i| = ||\Delta\texttt{cost}(\text{geomtric vars.})||_\infty \tag{5}$$

The first order optimality drops from $7.45 \times 10^{-9}$ to $1.33 \times 10^{-15}$ after scaling, indicating an extremely effective scaling.

## 1.2 Efficiency and General Optimization Considerations

When it comes to efficiency and power output, things become complicated. If the optimum solution is taken as the one that optimizes efficiency, then the situation is very different. The optimization algorithm is run anew without scaling, and the resultant Hessian[1] at the optimum, is given by,

$$\mathbf{H}_{\mathbf{x_{opt}}} = \begin{bmatrix} 4.891 & -4.1653 & 1.150 & 0.0 & -2.3737 \\ -4.1653 & 9.9191 & -1.1196 & -0.0 & 6.2427 \\ 1.1501 & -1.1196 & 0.8515 & 0 & -0.5468 \\ 0.0 & 0.0 & 0.0 & 1 & 0.0 \\ -2.373 & 6.2427 & -0.5468 & 0.0 & 3.972 \end{bmatrix} \tag{6}$$

[1] that is, the numerical approximation for it using MATLAB's built-in function (FD)

- computed using finite differencing. This looks fairly innocent, but it turns out that the condition number is extremely large. That is, $\kappa = 5041$. Also, since large and small numbers are so sporadically distributed in the matrix, there is no easy way of finding a good scaling. Rather, a systematic approach is undertaken. First, let us write up the new matrix in terms of the scaling parameters, $\mathbf{L}$. This is computed by taking first taking the outer product of $\mathbf{L}$ with itself; then taking the Hadamard product between the resulting matrix and $\mathbf{H}_{\mathbf{x_{opt}}}$ before the scaling. This yields

$$\hat{\mathbf{H}}_{\mathbf{x_{opt}}} = \begin{bmatrix} 4.891L_1^2 & -4.1653L_1L_2 & 1.150L_1L_3 & 0. & -2.3737L_5L_1 \\ -4.1653L_1L_2 & 9.9191L_2^2 & -1.1196L_3L_2 & -0. & 6.2427L_5L_2 \\ 1.1501L_1L_3 & -1.1196L_3L_2 & 0.8515L_3^2 & 0 & -0.5468L_5L_3 \\ 0 & 0 & 0 & L_4^2 & 0 \\ -2.373L_5L_1 & 6.2427L_5L_2 & -0.5468L_5L_3 & 0 & 3.972L_5^2 \end{bmatrix} \tag{7}$$

It should be noted that this approach is just a suggestion of our own, whose sole purpose is to bring down the computed condition number, at the given optimum. It builds on the Taylor approximation to second order, where the function is assumed to behave quadratically along the diagonal, and linear in both variables in the cross terms. Of course, there is no way of guaranteeing that the scaling actually translates into something that is well approximated by a Taylor series like that very far from the optimum[2]. Maybe, the actual objective function[3] could even have a sort of delta-function like behavior up until right before the optimum.

Anyway, for optimizing the condition number, a numerical approach is used where we loop through each component of $\mathbf{L}$ and optimize individually. The optimum is then fixed and used for computing the optimum of the next component, and so forth. There will no doubt be more intelligent ways to undertake a joint optimization of all the components and once[4], but this approach turns out to be extremely effective too. As every other component is held fixed, the condition number is plotted as the one in question varies. The optimum is then determined graphically. As nice as the graphs look, let us skip ahead to the results though.

[2] should of course always be possible right at it
[3] this is recognized as a sort of abstract idea, since no objective function is actually defined analytically
[4] somewhat ironically, we could have used an advanced optimizing algorithm to optimize this

The final scaling vector looks like this, $\mathbf{L} = [0.25; 0.6; 1; 0.15; 0.9]$. The condition number has thus been reduced from 5041 to 310. This is more impressive than it looks. Randomly varying components of the scaling vector was attempted too. This was never able to yield a condition number below 4000, so the systematic approach definitely proved itself useful. Looking at the diagonal elements, $\hat{\mathbf{H}}_{\mathbf{i,i}}(\mathbf{x_{opt}})$, there was not a lot to be gained. $\hat{\mathbf{H}}_{\mathbf{2,2}}(\mathbf{x_{opt}})$ and $\hat{\mathbf{H}}_{\mathbf{3,3}}(\mathbf{x_{opt}})$ did vary by an order of magnitude, but all are already $\sim O(1)$. Furthermore, it turned out that trying to re-scale without considering the condition number almost always increased it. Computation time only increased. All this may have seemed like overkill, as we are not being asked for it anyway. However, it turned out we were in dire need of good scaling as our problem is of a complexity which makes the computation time take hours otherwise. When rerunning with the scaling, the condition number of the new optimum had come all the way down to just 11.5. It converged with a time reduction of only 14% though, and the quality of the optimum deteriorated considerably, going from an efficiency of $3.15 \cdot 10{-}5$ to $8.8^{\cdot}10^{-6}$, in units of the scaling factor mentioned in report 3. This emphasizes an important point, to which we will return in *Multiobjective Optimization*; the robustness of the gradient-based method is extremely poor for our particular problem, unless settings are chosen which make it run much too slow.

Due to the extreme degree of non-linearity in our problem, local minima will likely be plentiful and *deep*. This owes partly to the fact that the modelling of the heat transfer is from the two-dimensional heat diffusion equation, where the heating time in worst case goes up exponentially to the power of 2, as the solution of the time and position dependent temperature is on a form *similar* to, $K(t, x, y) = \frac{1}{(4\pi t)^{d/2}} e^{-|x-y|^2/4t}$ [5].

First of all though, the magnetic aspect of the problem makes it extremely non-linear. The model has been programmed with adaptive evaluation boundaries, and when the width of the active material decreases, the magnetic flux[6] can become extremely small if the bounds on the variables allows for a more complete exploration. In our project, we were ambitious and allowed for the dimensions of the active material to change by two orders of magnitude

[5] The two-dimensional behavior will come into play when the width of the active material becomes small compared to the width of the yoke. Combined with a large thickness of active material, the power output becomes as good as 0

[6] to the **square** of which power is proportional

for most of the optimization we have done. As a first order approximation, that causes a change in the power output of a $10^4$ on a linear scale. Furthermore, this is just the obvious consequence of only wrapping our coil around a smaller rod; much more impactful, and unpredictable, is how the magnetic vector potential reacts to these changes. The curl will stay the same but how much flux ends up being guided around in the "magnetic circuit" is highly susceptible to change in any geometric parameter and extremely complicated from the perspective of the governing equations. The magnetic field is by nature divergence-less and rotational. It is also highly adaptive to changing geometry under the right permeability conditions, as it may condense many orders of magnitude.

Both power and efficiency depend on the magnetic simulation, and power furthermore depends on the heating time.

The actual scaling to be used in multi objective optimization took this as a starting point at was afterwards experimented with several times to get the best results for the *multiobjective optimization*. In the end, the fastest computed Pareto front and the best results came from running a multi-objective optimization based on a genetic algorithm, not a gradient-based one. This was implemented with no scaling at all. With such a high degree of non-linearity and unpredictable behavior, the genetic algorithm proved superior. As noted in report 3, it was slower, but the results were also less prone to producing spurious conclusions.

## 2 Multiobjective Optimization

Select two objective functions for your project

Multiobjective optimization turned out to be extremely troublesome. At first, the AWS and NBI methods were employed with SQP used for optimizing the weighted sum. This was done in the two-dimensional spaces of efficiency and power output, efficiency and cost, and power output and cost. We believe that both the AWS and NBI algorithms were successfully implemented. However, despite our best efforts to tune these algorithms, we were unable to get them to produce reasonable results within a reasonable time-frame and the algorithms

took extraordinarily long times to run. We largely attribute this difficulty to the complicated nature of our COMSOL-coupled objectives. The team concluded that MATLAB's fmmincon function and all of its algorithmic settings are poorly suited for our problem.

Multiobjective optimization of power output and cost turned out to be especially problematic. This could perhaps have been predicted from the start, although we must admit that we were very optimistic at first. There are two main reasons for this, as we see it:

- Both objectives are extremely non-linear and the combination of them, even more so.
- Both objectives are largely co-directional up until a level of *hard-to-resolve finesse*, and co-dependent by a myriad of mechanisms

The latter requires a bit of explanation. There are a few common features that will both enhance the power output and the efficiency. As has been explained in *report 2*, both of them scale with the total magnetic flux squared and will thus benefit from increased height of the active material. They will also both benefit from having a larger permanent magnet. As long as we can change variables and improve both objectives however, we are simply not at Pareto optimum. Getting to the point where one of them will have to hurt the other turns out to be a matter of finesse which was too much for simple optimization tools to be able to resolve to a level of[7].

Consider on the other hand yoke widths that are large. Then the heating time and the heat input will be increased, as the heat diffusion becomes increasingly two-dimensional in nature and more "waste material" has to be heated up alongside the active material. At the same time, this will mean that there is less fringing field, so that more magnetic flux can be guided through the coil. Less fringing leads to more power output, which leads to more efficiency. At the same time, increased heating time leads to *less* power output and increased heat input leads to *less* efficiency. This is just taking the variation of one variable into consideration and considering only the obvious effects, and *some* of the feedback mechanisms. The optimum where the sign of the effect on one of the objectives differs from that one the other will come down to the exact nature of the non-linearities and will be highly sensitive

[7] this is mainly a critique of *fmincon*

to the state of all the other variables.

Neither NBI or AWS could produce reliable results for multiobjective optimization of power and efficiency. We know that both of these methods do have their limitations and can produce erroneous results. Combined with the tendency of $fmincon$ to produce spurious results for these particular objective functions, it is not a huge surprise. AWS was implemented with some success for optimizing efficiency and cost jointly. It was painfully slow though, and a full and evenly space Pareto front will only be available the morning after the deadline of this report.

The team turned back to revisit the use of a genetic algorithm, which had previously been discarded due to its unreasonably large compute time. The number of function evaluations needed to be reduced for a genetic algorithm to suit our needs. So, the team turned to MATLAB's documentation to understand why the GA performed more function evaluations than there were population members in each generation. We found that while the GA handles linear constraints without trouble, nonlinear constraints rapidly increase the computational cost of the algorithm. Where one function evaluation takes place per population member in each generation with linear constraints, nonlinear constraints "confuse" the algorithm and require many more function evaluations per individual. The constraints related to length and width of our thermomagnetic generator are in fact already linear, but had been represented as nonlinear in MATLAB previously. The volume constraint is nonlinear. For the sake of experimentation, we relaxed the volume constraint entirely to explore the GA's performance. We may linearize the volume constraint later on, or remove it as we feel it may be unimportant in the scheme of our problem. Although we were able to make fmincon work for single objective optimizations, the success largely came from our ability to run the sqp algorithm many times and cherry pick the best results. To further reduce computing cost, we reduced the resolution of the mesh for both the thermal and the magnetic COMSOL model. During this process we were careful to not make the mesh so coarse that our objective functions returned inaccurate results.

The genetic algorithm applied with only linear constraints quickly showed promising results in a timely manner. Where previously we had observed poor results with runs lasting

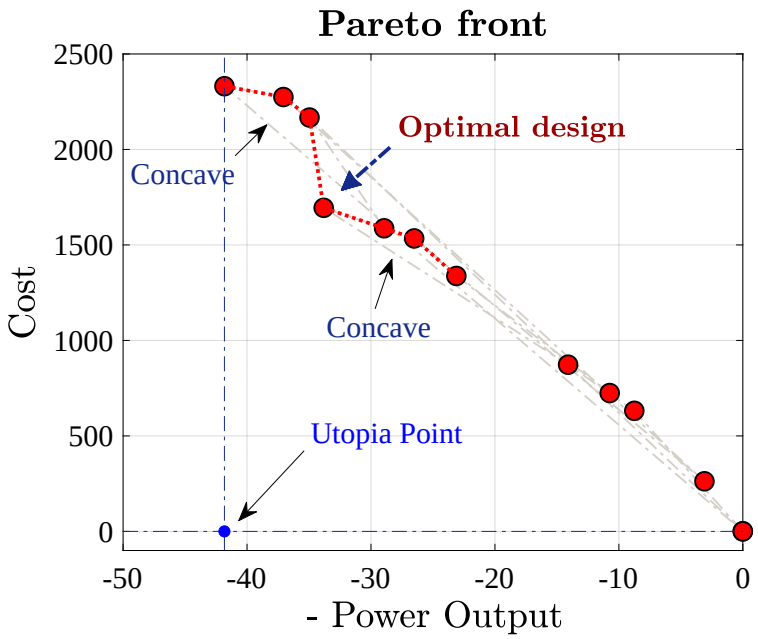


Figure 1: The Pareto front for optimizing both the cost and power output module. This is exactly what we would expect. As cost is being minimized with an eye for power too, there will be a common interest in removing active material from the structure, which is both the most expensive and results in exponentially larger heating times (exponentially smaller power outputs), but as power output becomes dominantly weighed, it will be more advantageous to morph active material into a horizontally elongated structure, rather than removing it, which hurts the magnetic flux and thus the power.

24 hours or more, we now could achieve reasonable results in less than 30 minutes. We first confirmed with single objective optimizations that the genetic algorithm offered comparable performance to fmincon with greater consistency. Then, we applied MATLAB's gamultiobj() function to our cost and power objectives and plot the resulting Pareto front in Figure 1.

We see that while cost is mostly linear with power, the optimal design does lie on a convex point and corresponds to a final selected design vector of:

| | |
|---|---|
| Yoke Width | 0.0709 |
| Yoke Height | 0.3625 |
| Permanent Magnet Height | 0.1464 |
| Active Material Height | 0.1499 |
| Gap Width | 0.1313 |

And gives the results:

$$\text{Cost} = \$1{,}694.4$$

$$\text{Power} = 33.8145$$

The optimal point marked in Figure 1 This convex point could potentially be a result of a high performing outlier from the optimization. However, we believe that the Pareto front is likely accurate because the algorithm performed more than 5000 function evaluations on a constant population of 35 individuals to gather the data. Had we realized that the algorithm would run for so long, we would have reduced the function tolerance and increased the population size to create a higher resolution curve. We plan to do this for the final presentation and report. Note that while the cost axis represents accurate dollar amounts, the power axis is on an arbitrary scale because per the setup of our problem, power is multiplied by an unknown constant K. Note also that power is negative. This is because the algorithm tries to minimize the objective, but we want to maximize power. Thus, we can multiply power by -1 and represent maximum power by optimizing for the most negative objective value. This approach is also taken with the efficiency objective. The following settings were used to create Figure 1:

```
funcTol = 1e-4;
conTol = 1e-5;
popSize = 35;
crossoverRatio = 1.2;
crossoverFraction = .8;
maxStallGenerations = 50;
```

For `gamultiobj()`, the algorithm is set to stop when the geometric average change in Pareto spread across all generations is less than the function tolerance divided by maximum stall generations. The above settings resulted in an impractically long computation time, and the algorithm was halted after approximately eight hours and the data was used to create the Pareto front in Figure 1.

It is an easy matter to take a point and show that it is non-dominated, and luckily, this can be shown for every single point on our Pareto front[8]. The chosen point is marked in figure 2. This figure contains the explanation as to why the point is no-dominated in its caption.

Using what we learned from the dual-objective optimization of cost and power, the team revised the MATLAB scripts to create a Pareto front for all three objectives: power, cost, and efficiency. The script and all helper functions used to create the three-objective Pareto front are included in the Appendix. The same script was used with a modified objective

[8]- meaning that is actually a Pareto front

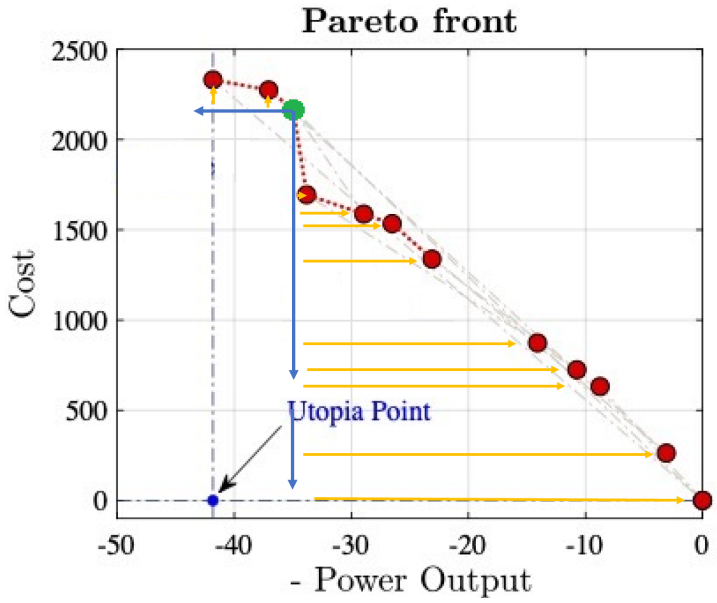


Figure 2: Consider the green point on the Pareto front. This is clearly non-dominated, as can be shown by drawing the blue arrows in the direction of decreasing cost and increasing power output. If we go along that line, then we will only be able to go to points that have also increased in cost, or decreased in power, respectively. The green point will thus always be optimal, as long as the particular weighing used to produce it, is considered. For a particular niche in terms of weighing, no other point can beat it.

function to produce the two-objective optimization. The 3D Pareto front produced is shown in Figure 3 and appears to be consistent with the 2D power-cost Pareto front when viewed from its side. We attempted to rush the completion of this optimization and thus the result does not show enough points along the Pareto front to draw a clear optimal line. The team is excited to run the algorithm again with a higher population and lower tolerances to produce a more complete graph for the final report and presentation.

## Contributions

### HW3:

- **Q1**: Mads: Writing & COMSOL/MATLAB integration, Problem formulation; Will: COMSOL/MATLAB integration
- **Q2**: Hanfeng: Writing & graph, Heuristic algorithm, GA programming; Will: programming that worked (Single objective GA), results generation, analysis and writing.
- **Q3**: Will: Main programming (cost, power output, and efficiency), figures generation (running from main program) results generation, & revision. Mads: Results analysis,

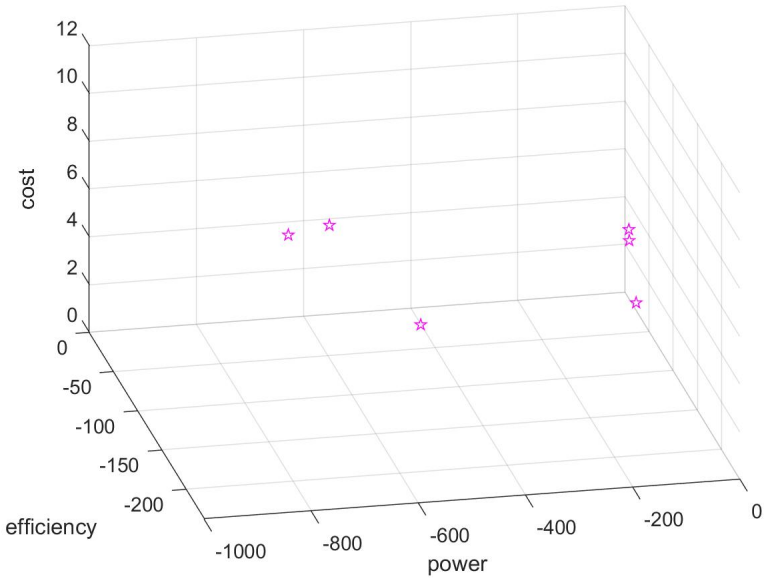


Figure 3: 3D Pareto front of power, efficiency, and cost. This was created with a rushed algorithm and did not produce enough data points to be meaningful enough to draw conclusions from.

and writing. Hanfeng: Writing, programming & analysis for sensitivity analysis & single objective optimization for cost, & graphic representation (schematics).

- **General**: Hanfeng: Beautiful figures (added by Will) & latex formatting. Putting up website with results and project description

**HW4**:

- **Q1**: Hanfeng: Scaling for cost module & writing. Will: Programming for both modules, editing & revising. Mads: scaling for the efficiency module & writing.
- **Q2**: Will: Programming NBI, programming MOO GA, optimization running, results analysis, writing. Mads: Programming, results analysis, writing. Hanfeng: Schematic & Graphic representation, revision.
- **General**: Mads: Discussion on optimization considerations, SQP AWS programming. Will: Extremely labour intensive and productive programming (added by Mads). Hanfeng: Latex formatting & schematics and the figures

## Appendix

**TherMaG_gradoptim_L_single_scaling.m**: Script for exploring design variable scaling of different objective functions.

```
clear
clc
close all

% starting point for optimization
x0 = [.05;.1;.05;.05;.05]; % default
x0cost = [.15;.15;.05;.05;.05]; % cost
x0power = [.05;.45;.25;.15;.15]; % power
x0efficiency = [.05;.45;.25;.15;.05]; % efficiency

TolF = 1e-6;
TolX = 1e-6;
TolCon = 1e-6;

% Set nondefault solver options
options = optimoptions('fmincon','Algorithm','sqp','FunctionTolerance',TolF,'StepTolerance',
    TolX,'ConstraintTolerance',TolCon,'PlotFcn',{@optimplotfval, @optimplotfunccount});

% set upper and lower bounds
lb = [.001, .001, .001, .001, .001];
ub = [.5, .5, .5, .5, .5];

% make L scaling vectors
L = [1; 1; 1; 1; 1];

% make anonymous functions
powerFcn = @(input) powerFcnSingleL(input, L);
efficiencyFcn = @(input) efficiencyFcnSingleL(input, L);
costFcn = @(cost) objee_scaling(cost, L);

% constants
V_max = .125;
L_max = .5;

% set linear constraints
A = [0, 1, 0, 0, 0; 2, 0, 0, 0, 1; 0, -1, 1, 1, 0];
b = [L_max; L_max; 0];

tic;

```

```
% run the optimization
% CHANGE THIS TO THE CORRECT VARIABLE
[solution,objectiveValue,exitflag,output,lambda,grad,hessian] = fmincon(costFcn,x0,A,b
    ,[],[],lb,ub,[],options);

unscaledTime = toc;

%% Run the above section once, then only run this section to check new scaling vectors

L_efficiency = [1; 10^(-3.5); 10^(-4); 1; 10^(-3.5)];
L_power = [1; 1; 10^(-.5); 1; 1];
L_cost = [10^(-1.5); 10^(-1); 1; 1; 1];

fprintf('No scaling: The best solution is efficiency = %.8f, corresponding to a design
    vector of: \n',objectiveValue);
fprintf('\n');
disp(solution);
fprintf('\n');
fprintf('This solution had the following simulation output info: \n');
fprintf('\n');
disp(output);

eigs = eig(hessian);
maxeig = max(eigs);
mineig = min(eigs);
conditionNumber = abs(maxeig/mineig);

fprintf('The unscaled hessian is:\n');
fprintf('\n');
disp(hessian);
fprintf('The unscaled condition number is %d',conditionNumber);

powerFcn = @(input) powerFcnSingleL(input, L_power);
efficiencyFcn = @(input) efficiencyFcnSingleL(input, L_efficiency);
costFcn = @(cost) objee_scaling(cost, L_cost);

tic;

% run the optimization again with the new scaling
% CHANGE THE FUNCTION TO THE CORRECT VARIABLE
[scaledSolution,scaledObjectiveValue,scaledExitflag,scaledOutput,scaledLambda,scaledGrad,
    scaledHessian] = fmincon(costFcn,x0,A,b,[],[],lb,ub,[],options);

scaledTime = toc;

[realScaledPower, realScaledEfficiency] = efficiency_power_modules(scaledSolution);
```

```
realScaledCost = objee(scaledSolution);

% CHANGE THIS TO THE RIGHT VARIABLE
realObjectiveValue = realScaledCost;

% CHANGE THIS TEXT TO THE RIGHT VARIABLE
fprintf('Efficiency scaling: The best solution is efficiency = %.8f, corresponding to a
    design vector of: \n',realObjectiveValue);
fprintf('\n');
disp(scaledSolution);
fprintf('\n');
fprintf('This solution had the following simulation output info: \n');
fprintf('\n');
disp(scaledOutput);

eigs = eig(scaledHessian);
maxeig = max(eigs);
mineig = min(eigs);
scaledConditionNumber = abs(maxeig/mineig);

funcCountDiff = -(scaledOutput.funcCount - output.funcCount)/output.funcCount*100;
fvalDiff = -(realObjectiveValue - objectiveValue)/objectiveValue*100;
timeDiff = -(scaledTime - unscaledTime)/unscaledTime*100;

fprintf('The scaled hessian is:\n');
fprintf('\n');
disp(scaledHessian);
fprintf('The scaled condition number is %d',scaledConditionNumber);
fprintf('\n');
fprintf('The scaled optimization found a %.2f percent smaller optimal value in the same
    number of iterations and %.0f percent fewer function evaluations\n',fvalDiff,
    funcCountDiff);
fprintf('\n');
fprintf('The scaled optimization found a solution %.2f percent faster than the unscaled
    optimization',timeDiff);
```

**powerFcnSingleL.m**: Helper function to return power.

```
function [power] = powerFcnSingleL(x,L)
    [power,~] = efficiency_power_modules_L_single_scaling(x,L);
end
```

**efficiencyFcnScaling.m**: Helper function to return efficiency.

```
function [efficiency] = efficiencyFcnSingleL(x,L)
    [~,efficiency] = efficiency_power_modules_L_single_scaling(x,L);
end
```

**objee.m**: The scaled objective function for running minimizing the cost of the TMG.

```
function f = objee(x)
% x = [h_A; w_gap; w_yk; h_yk; h_pm];
  Gadolinium = 1.71553e5; % $/m^3
  Iron = 1.4804e3; % $/m^3
  Neodymium = 1.628e5; % $/m^3
  w_yk = 10^(-7)*x(1);%10^(-7)*
  h_yk = 10^(1)*x(2);%10^(1)*
  h_pm = x(3);%1*
  h_A = 10^(1)*x(4);%10^(1)*
  w_gap = 10^(-10)*x(5);%10^(-10)*
  magnetArea = h_pm*w_gap;
  activeMaterialArea = h_A*w_gap; % in reality, this would not be a solid mass
  yokeArea = 2*w_yk*h_yk;
  f = Gadolinium*activeMaterialArea + Iron*yokeArea + Neodymium*magnetArea;
end
```

**geocon.m**: The geometric constraint for running minimizing the cost of the TMG.

```
function [c,ceq] = geocon(x0)
    x = x0;
    inputmatrx = x;
    h_A = inputmatrx(1);w_gap = inputmatrx(2);
    w_yk = inputmatrx(3);h_yk=inputmatrx(4);h_pm=inputmatrx(5);

    c(1) = h_yk*(2*w_yk + w_gap) - .125;
    c(2) = h_yk - .5;
    c(3) = (2*w_yk + w_gap) - .5;
    c(4) = (h_A + h_pm) - h_yk;
    ceq = [];
end
```

**objee.m**:

The unscaled objective function for running minimizing the cost of the TMG.

```
function f = objee(x)

    Gadolinium = 1.71553e5; % $/m^3
    Iron = 1.4804e3; % $/m^3
    Neodymium = 1.628e5; % $/m^3

    w_yk = x(1);
    h_yk = x(2);
    h_pm = x(3);
    h_A = x(4);
    w_gap = x(5);

```

```
    magnetArea = h_pm*w_gap;
    activeMaterialArea = h_A*w_gap; % in reality, this would not be a solid mass
    yokeArea = 2*w_yk*h_yk;
    f = Gadolinium*activeMaterialArea + Iron*yokeArea + Neodymium*magnetArea;
end
```

**TherMaG_GA_Multi.m**: The script used to run multiobjective genetic algorithm optimizations.

```
clear
clc
close all

% set upper and lower bounds
lb = [.001, .001, .001, .001, .001];
ub = [.5, .5, .5, .5, .5];

% set optimization options
funcTol = 10^(-2.5);
conTol = 1e-5;
popSize = 30;
crossoverRatio = 1.2;
crossoverFraction = .8;
maxStallGenerations = 5;

options = optimoptions('gamultiobj', ...
        'CrossoverFcn',{@crossoverheuristic,crossoverRatio},...
        'FunctionTolerance', funcTol, ...
        'PopulationSize', popSize, ...
        'CrossoverFraction', crossoverFraction,...
        'ConstraintTolerance', conTol,...
        'MutationFcn',{@mutationadaptfeasible},...
        'MaxStallGenerations', maxStallGenerations,...
        "PlotFcn", {@gaplotPareto, @gaplotgenealogy, @gaplotscorediversity, @gaplotrankhist,
    @gaplotstopping},...
        'Display','diagnose');

% constants
V_max = .125;
L_max = .5;

% set linear constraints
A = [0, 1, 0, 0, 0; 2, 0, 0, 0, 1; 0, -1, 1, 1, 0];
b = [L_max; L_max; 0];

% run the GA
```

```
[solution, fval, exitflag, output, population, scores] = gamultiobj(
    @powerCostEfficiencyMulti,5,A,b,[],[],lb,ub,options);
```

**powerCostEfficiencyMulti.m**: A helper function returning all three objective values for optimization.

```
function [evaluation] = powerCostEfficiencyMulti(input)

    [power, efficiency] = efficiency_power_modules(input);

    evaluation = zeros(3,1);
    evaluation(1) = power;
    evaluation(2) = efficiency;
    evaluation(2) = objee(input);

end
```

**efficiency_power_modules.m**: The objective function coupled with COMSOL to return power and efficiency.

```

function [power, efficiency] = efficiency_power_modules(vector)

    % define constants
    Cp_A = 450;
    Cp_everythingElse = 3.5433e6;

    % extract design parameters
    w_yk = vector(1);
    h_yk = vector(2);
    h_pm = vector(3);
    h_A = vector(4);
    w_gap = vector(5);

    % run models to get outputs for result calculations

    % open the thermal comsol model and run the study
    thermalModel = mphopen('therm_assign_3.mph');

    % set basic thermal model parameters
    thermalModel.param.set('yoke_width',w_yk);
    thermalModel.param.set('permmag_height',h_pm);
    thermalModel.param.set('activemat_height',h_A);
    thermalModel.param.set('yoke_height',h_yk);
    thermalModel.param.set('actperm_width',w_gap);

```

```
    % run the thermal model
    thermalModel.study('std2').run;

    % extract the solution (end of the time array)
    time = thermalModel.result.numerical('pev1').getReal();
    heatTime = time(end);

    % run the thermal model to find integrated temperatures
    integratedTemperaturesActive = thermalModel.result.numerical('int1').getReal();
    T_int_active = integratedTemperaturesActive(end);

    integratedTemperaturesEverythingElse = thermalModel.result.numerical('int2').getReal();
    T_int_everythingElse = integratedTemperaturesEverythingElse(end);

    % open the magnetic comsol model
    magneticModel = mphopen('simple_for_assign_3.mph');

    % set basic magnetic model parameters
    magneticModel.param.set('yoke_width',w_yk);
    magneticModel.param.set('permmag_height',h_pm);
    magneticModel.param.set('activemat_height',h_A);
    magneticModel.param.set('yoke_height',h_yk);
    magneticModel.param.set('actperm_width',w_gap);

    % compute flux before (mu = 1)
    magneticModel.param.set('mu_r',1);
    magneticModel.study('std1').run;
    fluxBefore = magneticModel.result.numerical('int1').getReal();

    % compute flux after (mu = 20)
    magneticModel.param.set('mu_r',20);
    magneticModel.study('std1').run;
    fluxAfter = magneticModel.result.numerical('int1').getReal();

    % calculate results

    % compute power result
    deltaFlux = fluxBefore - fluxAfter;
    power = -deltaFlux^2/heatTime*1e8;

    % compute efficiency
    efficiency = -deltaFlux^2/(Cp_A*T_int_active + Cp_everythingElse*T_int_everythingElse)*1
    e11;

end
```

## References

[1] Katoch, S., Chauhan, S.S. & Kumar, V. A review on genetic algorithm: past, present, and future. Multimed Tools Appl 80, 8091–8126 (2021).

[2] Tabak, Daniel; Kuo, Benjamin C. (1971). Optimal Control by Mathematical Programming. Englewood Cliffs, NJ: Prentice-Hall. pp. 19–20. ISBN 0-13-638106-5.

[3] fmincon Documentation. Mathworks, Inc. URL: https://www.mathworks.com/help/optim/ug/fmincon.html#busp5fq-7

[4] First-Order Optimality Measure. Mathworks, Inc. URL: https://www.mathworks.com/help/optim/ug/first-order-optimality-measure.html

[5] Professor Walter Murray, Systems Optimization Laboratory. Advanced Methods in Numerical Optimization. URL: https://web.stanford.edu/class/msande312/restricted/OPTconditions.pdf